Performance Analysis of Hashes and Encryption Schemes

Chesleah F. Kribs and Samir J. Yuja

Florida State University

# **INTRODUCTION**

To gain intuition and practical experience with encryption, we decided to implement several cryptographic modules and measure their performance. We measured the time it takes to hash, encrypt, and decrypt using different ciphers while using varying key sizes and file sizes. Along the way we learned: additional ciphers, the importance of choosing compatible components (keys, IVs, and modes of operation), became familiar with a cryptography library, and learned basic statistical analysis. It became a valuable experience we hope to share with our classmates.

# CODE IMPLEMENTATION

## SETUP

We implemented our modules in Python 2.7 using functions from the Cryptography Library 1.6. To obtain sample plaintext files of different sizes we concatenated all of Shakespeare’s works obtained from an online MIT course3. For example, to create the 10MB file we simply concatenated the 5 MB original text file twice using the command ‘cat shakes.txt >> 10\_MB.txt”. To run the modules, we ran the command “python module.py”, where “module.py” is one of the three provided modules. To obtain our timings; we manually entered input into these modules, and our analysis came from the results saved into an excel spreadsheet.

## MODULES

The modules were implemented using functional programming paradigm with several global variables; each menu, each encryption, each decryption is contained within individual functions within a module. There are three modules: “hash.py” for hashing, “asym\_encrypt.py” for asymmetric encryption/decryption and finally “sym\_encrypt” for symmetric encryption and decryption.  Each module is very similar, it presents a menu, option for input and other various methods. The first option it presents is a menu to the user. These menus take a user’s choice as input, then other methods validate the user input. Finally, encryption or decryption is carried out using cryptography library methods.

Note: Due to a design choice, encryption and decryption must be performed in the same run of the module. When using encryption, we save the encryption object to an array, to later obtain a decryption object from it. We decided to confine encryption and the corresponding decryption to a single run, since this serves our purpose well. To illustrate, we would encrypt a 10MB file with AES using CBC and obtain a timing; then in the same run, we would decrypt that file using a decryption object obtained from the corresponding encryption object. We could encrypt using different encryptions, but to decrypt them the corresponding encryption object must have been saved in the global array.

## **HASH.PY**

See Appendix A.

## **ASYM\_ENCRYPT.PY**

See Appendix B.

## **SYM\_ENCRYPT.PY**

See Appendix C.

# INITIAL HYPOTHESES

Since we are using multiple variations to determine performance analysis we hypothesized for each variation of hashing, symmetric encryption/decryption, and asymmetric encryption/decryption. We came up with initial hypotheses that related the key size and file size to the time required to hash or encrypt/decrypt. Using our hypotheses, we could organize our code in a way we could test. For hashing, we hypothesized that the time required to hash will grow linearly with the specified file size. For symmetric encryption and decryption, the relationship between the file size and the time take we believe will be linear. Also, since the relationship we estimate is linear we know that y = mx + b, and when we increase the key size we estimate that the linear relationship will shift up by such constant key size, b. In regards to asymmetric encryption and decryption, we estimate that it will be much slower than hashing and symmetric encryption and decryption. Specifically, RSA will grow exponentially, since there is a large modular exponentiation. We estimated this because the public key exponent, e, is typically very small, but the private key exponent d is very large. We think it is important to note that other encryption schemes like DSA and ECC use hashing within encryption, therefore these don’t encrypt or decrypt anything very large. DSA and ECC are mostly used for signing, and therefore hashing before signing would reduce their workload.

# ALGORITHMS NOT DISCUSSED

We implemented a few hash functions and symmetric encryption schemes that were not seen in class lectures. These hashes and encryption schemes include: RIPEMD, Whirlpool, Camellia, CAST5/CAST-128, and SEED. The basic explanations for a hash function and symmetric encryptions, as well as the specific methods used, can be found below.

## HASH FUNCTION:

A hash function accepts a message m and produces a message digest of a fixed length. It is a kind of ‘signature’ for a stream of data. Hash functions are not to be confused with encryption, as encryption transforms data from plaintext to ciphertext and can be decrypted, otherwise known as a two-way operation. However, a hash is a strict one way operation, and consists of the same size no matter the size of the input. A hash function’s main goal is to ensure message integrity. The requirements of a basic hash, given a function h: X-> Y, h is:

1. Preimage resistant (a one-way function): given any y in Y, it is computationally infeasible to find a value x, in X, such that the hash of X is y.
2. 2nd Pre-image resistant (weak collision resistant): given x in X, it is computationally infeasible to find a value of x’ in X such that x’ is not equal to x and the hash of x’ is equal to the hash of x.
3. Collision Resistant (strong collision resistant): if it is computationally infeasible to find two distinct values x’ and x in X such that the hash of x’ is equal to the hash of x.

RIPEMD-160 / RIPEMD Hash Function

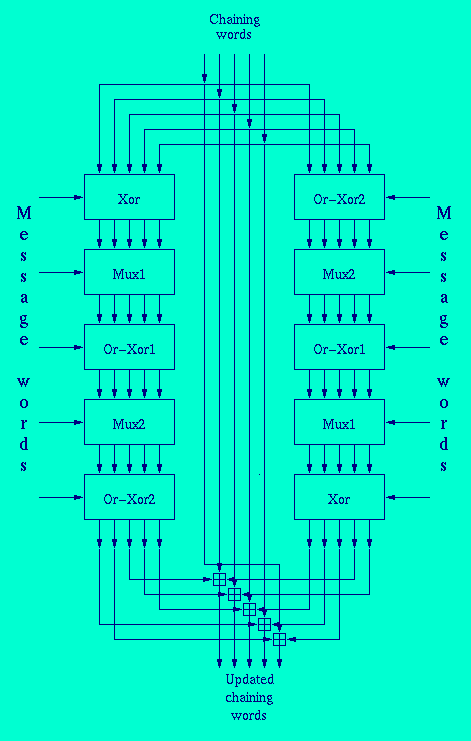
An algorithm we implemented was the RIPEMD160 (“RIPEMD”) hash. This hash is a 160-bit hash designed by Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. The name itself represent the acronym: RACE Integrity Primitives Evaluation Message Digest. RIPEMD160’s predecessor, RIPEMD, was first published in 1996 and developed at the COSIC research group. The design principle was based on MD4, but with similar performance to SHA-1. The python code implemented in our project was that of RIPEMD160, which is an improved 160-bit version and the most common of the RIPEMD family. It was developed in the open academic community in contrast to NSA developed SHA-1 and SHA-2 it compares itself to. Other variations of the family include: RIPEMD-128, RIPEMD-256, and RIPEMD-320. The latter two versions, do not increase the security (pre-image attacks) in respect to the 128-bit or 160-bit versions. Since the original RIPEMD was structured after MD4, RIPEMD was less efficient than MD5. However, with the design of the 160-bit, it supplemented these problems; but, with limited results premiering after SHA-1, and being slower than SHA-1. For the exposure and analysis, RIPEMD-160 seems robust but with minimal performance. Our recommendation is to stick with the standard SHA-256 as it’ll withstand many of the attacks that RIPEMD-160 is subjected to.

Figure 1: RIPEMD hash

### Whirlpool Hash Function

Another hash function not reviewed in class was the implementation of the Whirlpool hash. Whirlpool is a block-cipher-based secure hashing function. It was developed by Vincent Rijmen, a co-inventor of Rinjdael, and Paulo Barreto. Whirlpool is one of only two hashing functions endorsed by NESSIE (New European Schemes for Signatures, Integrity, and Encryption). It was created to match the performance and security, if not to exceed, similar block-cipher based hash functions like SHA. Its’ underlying block cipher is based on AES, with the security goals including an expected workload of generating a collision to the order of 2n/2 executions and given an n-bit the computation of finding a message to hash of that value is 2n.

The implementation of Whirlpool takes a message as input with a max length of less than 2256 bits and produces a 512 bit MD. The message is padded so that the length is an odd multiple of 256 bits, and padding is always added in the range of 1-512 bits. A block of 256 bits is then appended, and the outcome of the first two steps goes into the block cipher W, explained next. The block cipher takes in a 512-bit block of plaintext and a 512-bit key as input and produces a 512-bit block of ciphertext as output. It uses four different functions: add key (AK), substitute bytes (SB), shift columns (SC), and mix rows (MR). W consists of a single AK followed by 10 rounds with all four functions: RF(Kr) = AK[Kr] ∘ MR ∘ SC ∘ SB, where Kr is the round key matrix for round r. The overall W block cipher algorithm can be described as: W(K) = (O 10, r = 1, RF(Kr)) ∘ AK(K0). The performance of Whirlpool is good, with little implementation experience, but since Whirlpool uses the same functional building blocks as AES we can expect similar performance. In comparison with SHA-512, Whirlpool requires more hardware resources but outperforms SHA-512. The potential drawbacks are that of block ciphers themselves. They exhibit regularities and weaknesses such as slow performance than hashes based on compression functions. Block Cipher hashes have a hash length equal to twice the size of the cipher block length, which has usually been limited, like in DES or 3DES.

## **SYMMETRIC ENRYPTION**

### CAMELLIA

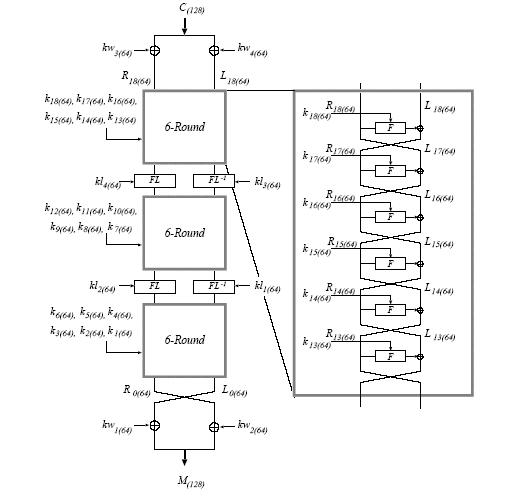
Camellia is a symmetric key block cipher developed jointly in 2000 by researchers at Mitsubishi and Nippon Telegraph and Telephone (NTT). Camellia specifies 128-bit block size and has the same interface as AES. The advantages of such a system include that it is designed to enable flexibility in both hardware and software implementations on 32-bit processors used in smart cards, crypto hardware and embedded systems, furthermore its key setup time is superb and its key agility is superior to AES.

Figure 2: Camellia Symmetric Encryption

Camellia is a Feistel cipher with 18 rounds in a 128-bit key or 24 rounds when using 192 or 256-bit keys. Every six rounds, a transformation layer is applied (the FL-function) or the inverse. It uses four 8x8 bit S-Boxes with input and output and logical operations, and it also uses input and output key whitening. It is considered a modern safe cipher, as its considered infeasible to break it by brute-force attack with current technology. However, it has been scrutinized by the cryptographic community, most recently being selected as a recommended cryptographic primitive by EU NESSIE and being subjected in Japan’s e-government systems which were selected by Japan’s Cryptography Research and Evaluation Committees (CRYPTREC). Unfortunately, Camellia is patented and thus individuals are a bit wary about a patented algorithm, due to much uncertainty. It is possible that many cryptography library developers are less likely to implement Camellia because of this issue. Since Camellia is newer, the benefits are not as well-known as its complement in AES.

### CAST5/CAST-128

CAST5 or CAST-128 is a block cipher most notably used in PGP. It was created in ’96 by Carlisle Adams and Stafford Tavares using the CAST design protocol, it has been mentioned that the name CAST was created using the first and last names of the authors. The design of CAST5 is a 12- or 16-round Feistel structure like the Camellia Block Cipher (which uses 18- or 24-rounds). It has a key size between 40 and 128 bits, but only in 8 bit increments. The full 16 rounds are used when the key size is longer than 80 bits. CAST-128 uses a rotation to provide immunity to linear and differential attacks; it uses XOR, addition, and subtraction (mod 2\*\*32) in the round function, and uses three variations of the round function itself throughout the cipher. Because of its immunity to linear or differential analysis (Unlike DES), CAST5 creates a method that gives values for rounds, r, that make differential cryptanalysis impractical or impossible.

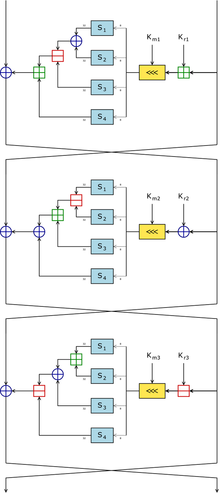
[](https://en.wikipedia.org/wiki/File:CAST-128-large.png)The model for CAST5 or CAST-128 has 8\*32 S-boxes, four in the key schedule and four in the encryption, while the round keys are 37 bits. The F functions XORs the input with 32 bits of round key, splits the result into bytes and runs each byte through a different S-box to get four 32-bit results. They are combined non-linearly, using different combining functions in different rounds. Then, the output is given a rotation controlled by the other five round key bits. CAST5 most serious (but still by mildly) weakness is its small block size (64-bits) by modern standard, like AES’ 128-bit. It might turn to be a problem, although for two decades it hasn’t, but the increasing probability for collisions among random blocks occur when enciphering large amounts of data with the same key. It may also be vulnerable to a timing attack related to cache or data-dependent operations.

Figure 3: CAST5/CAST-128

### SEED

SEED is a block cipher developed by the Korea Information Security Agency (KISA). It is seldom found anywhere besides South Korea, but gained popularity because of the development of KISA’s own standard outside of the 40-bit encryption methods. The design of SEED is a 16-round Feistel structure with 128-bit blocks, like AES. It uses two 8x8 S-Boxes, which are derived by discrete exponentiation. The 128-bit full cipher is a Feistel with a F-function operating on 64-bit halves, while the F-function itself is a Feistel structure composed of a G function operating on 32-bit halves. But, the recursion does not extend because the G-function is not a Feistel structure. In the G-function, the 32-bit message digest is considered as four 8-bit bytes, each of which are passed through one of the S-boxes, then combined through a set of Boolean functions which each output bit depends on the 3 of the 4 input bytes. SEED is robust against the linear cryptanalysis and differential cryptanalysis and other related key attacks.

# CHALLENGES

## VALIDITY OF KEYS

While conducting our experiment to encrypt and time files of increasing size we encountered several challenges: validation of keys, padding, and elliptic curve cryptography. The first was choosing a valid combination of correct sizes of keys, initialization vectors (IV) size, and stream ciphers for symmetric encryption. This resulted in a plethora of possible combinations: 5 ciphers, 6 modes of operation, and approximately 6 different key sizes. Some algorithms required that the IV size equal the block size, this meant that we needed padding.

## PADDING

Secondly, a problem we ran into was that padding was not taken care of automatically. Padding had to be done on a line-by-line basis, since the library we used did not allow padding to be done on an entire file at once, as we used low-level cryptographic functions. These ‘primitive’ functions allowed for more flexibility within our experiment; as the high-level functions of the library did not allow us to choose different ciphers and modes of operation. Because of this, we needed the more ‘primitive’ functions to analyze and measure performance differences across different ciphers. Thus, padding was done before encryption and after decryption. Determining the order was challenging since the padder object was different from the unpadder object, but the two objects both required multiple calls to member functions: “update” and “finalize”.

## SIGNATURE VERIFICATION

Then there was a bug in our code that we decided to look away from, elliptic curve cryptography signatures. Signing a file with Elliptic Curve Cryptography resulted in a minimal but still significant error approximately fifty percent of the time. During this time, fifty percent the verification would result in a valid signature; however, the other fifty percent the verification would result in an invalid signature. We could have one signature, and it be verified correctly and then later be marked as invalid a short time later. We remarked that it was as if the process of verifying the signature once damaged or modified it when repeating. We need to explore the cause of this error further, but for our experiment, we simply started from scratch for every run of ECC.

# IMPROVEMENTS IN IMPLEMENTATION

For our analysis, we determined it would have been better to print the signatures and encrypted files using base64 encoding. Without using this, when we printed the signature it resulted in multiple lines of garbled symbols, since the print statement interpreted bytes as new lines and some it couldn’t even interpret. The same problem occurred with the encrypted files, which we believe could have been solved had we used base64 encoding. The encrypted files may have been more legible had we base64 encoded them before writing the encryption. After getting the results we had without using base64 encoding we can say we now understand the reasons behind using encoding and why it exists. Some reasons for us in the future to use encoding would have been: for output to stay on a single line, and two, for legible output that we could use to visually verify the randomness of the encryption given the same output. See below for an example when not using base64 encoding.

https://lh3.googleusercontent.com/XO3csyRoKhiCYAzvPSIJDCl76RXa-ON9uSHAOOWO14jdlAdl3VYYQYQL9RS3u2050VrsgAvRh6UHbo5T-dGLSenY8oa4GykC70XFF0d9YiyxLqPW_m1VEA3BQDOgRdtnuktRM1BT

# ANALYSIS

## INTRODUCTION

We used statistical tests introduced in a summer course, Computer Systems Performance Analysis (CIS 5930), under tutelage of a close friend and classmate, Manuel Hernandez. Many of the methods and concepts are beyond our grasp of statistics, but the methods provided simply required reuse of the already existing formulas. After our calculations were completed, we had Manuel verify our results. In what follows, we will attempt to explain the most relevant statistical concepts used in our analysis.

## DESCRIPTIVE ANALYSIS

Before we began the statistical analysis, we conducted a brief descriptive analysis. To ensure our accuracy, we took several measures to ensure that the timings were done consistently. These measures included: timing when the machine was always plugged in, applications were closed before timing, and the lines in the code were the same for timings. The machine was always plugged into a power source to ensure that the machine was running at the most optimal and highest CPU frequency for all measures. The applications, although obvious, were closed to ensure that there were no background processes to slow down the encryption/decryption or hashing processes. Additionally, we used the same python functions for all our timings, meaning we ended the timings in the same locations of code. In the except of the code, you will notice that for encryption we started timing before the opening of the file, the for loops, and stopped timing after writing to the file, at the end of the for loop. The decryption code does not vary in this specification either. Finally, we expect any artifacts in our timings to either be consistent across all measurements or to be detected by our error functions in our analysis.

Furthermore, our analysis uses the average of several timings. Two separate runs were made for each measurement. We believe that this method will produce more accurate numbers for measuring the timings. In addition to this method, we produced the timing difference using floats, 64 bits each, for more accurate precision. These time differences were placed into the spreadsheet to be checked for errors using several formulas.

## ARTIFACTS IN OUR DATA

Initially, we began running file encryption onto two different terminal tabs. On one tab, we would have one file running for the same encryption cipher and file size, whilst on the other have a different encryption for the same file size. Using this method, we noticed that the left tab would run significantly slower than the right tab. The right was faster from anywhere between two to thirty seconds. We hypothesized that the right tab must have a higher scheduling priority on the machine.

https://lh4.googleusercontent.com/wprWKjZqvV_pNsHEy16XBNa7RX3EKTUa_cVFZebn7oSDyA3yafcvxE3FZd_L0yy3iEdmRuS4BCIKlReFK8G6eKlm-JTDLdUGP-StkWgUI_43Sm9lJb83RNRZrrQmcopYIeU3bfV_

In the above diagram, you can see the timing for a CAST5 encryption using a file size of about 1GB (~1GB). The right tab produced a result of 92.61556 seconds, while the left tab produced a time of approximately 131.5444 seconds. We looked for differences in both terminal tabs, but generally the right tab was the quickest. To look for differences, we ensured that both tabs were using the same variables: a cipher (CAST5), key size (40 bits), and the same file size (~1GB), including other additional similarities. We eventually noticed the directories used by the tables were in different locations: home for the left tab, and desktop for the right tab. We moved both to identical directories in the Desktop and were able to reduce some time differences. However, after removing some of the differences in the directory locations, the right tab still processed faster by a slight of a few seconds. We assumed that the right tab gets more CPU time.

A second artifact was done after redoing our timings from noticing that the time differences for signature verification was given in integrals. We found this rather odd, so we examined the code. The difference we determined was in the format specifier that we used; we had a coding inconsistency! We were printing the time difference using ‘%d’ instead of ‘%f’. We had to retake many measurements over.

**https://lh6.googleusercontent.com/G8wj0kughbBIeIqMNmvwjDtaFqQlCnmLO0HSX4g9BVR355u1lTL4ruygimXgcuQgYdY_rLkAB0ypZfZ1MB7v3Adzs_rpMIswjB8CV1xVyFhF4vEz29z44KHwCuFFcZ1ZHbfMaN0M**

## STATISTICAL ANALYSIS

First and foremost, our analysis hinged on a single measurement called the coefficient of determination (R2). R2 provided a measure of how well-observed outcomes fit our hypothesized model, in our case it was a linear model. For example, assume that we predicted that as the file size increases, our independent variable, the time required to hash, which is our dependent variable, will increase linearly. We would then perform the calculations and obtain our R2 value. This value could be .999, which would correspond to 99.9% of the variation in the time required can be explained by the linear relationship with the file size.

In our analysis, our calculated threshold for R2 was 0.95. From our original hypothesis, we predicted a linear relationship, then formulated R2, and if it was above 0.95, then we concluded that the relationship is linear. Otherwise if the model did not seem to fit linearly, we tested an exponential model. We did this by taking the natural log (ln) of the time and plotting alongside the file size. Furthermore, if the exponential model did not fit, then we proceeded to test polynomial models, starting with a polynomial of degree two, squaring, and then a polynomial of degree three, cubing. Each time calculating the R2 and using the same threshold.

In our statistical analysis we understood which independent variables were more significant in affecting the dependent variable, time. We used a formula that calculated the standard error of estimate (SE), which indicates for a given independent variable, the average distance from the regression line2. This would mean that for a smaller SE, the stronger the variable fits the linear regression. For example, the hash timings had a SE equal to 1.432E-8, and thus, from our understanding, this means that file size fits our linear model and that it is a strong predictor of the independent variable, time taken for hashing.

# CONCLUSION

All the hashes we tested are linear. When we plotted them using a linear model, we obtained very high coefficients of determination, above 0.99. This indicates that a linear model fits those well. We had no need to try other models (polynomial and exponential). Whirlpool was the slowest hash. We determined the slope, and it was the largest of all of them by an order of magnitude. The y-intercept was also the largest. This makes sense since timing was 3-5 seconds longer for each file size than for the other hashes. We believe that it is slowest perhaps because it produces the largest message digest, 512-bits. Surprisingly, the slowest hash per bit produced was MD5-128; it takes approximately 0.174 seconds per bit. Whirlpool was the fastest per bit followed by SHA-256.

The first thing we noticed about our symmetric key ciphers is that they work much faster when decrypting. Upon further investigation, we realized that this is due to CBC, cipher block chaining, and not the ciphers themselves. CBC runs sequentially for encrypting, but in parallel for decrypting. For faster encryption, use CTR, since CTR can be parallelized for both encryption and decryption. Clearly, the mode of operation has a significant impact on performance. We believe that the decrypting may run faster since the files have been cached after encrypting them. Computer architectures cache data that was recently used, temporal locality, into to increase read and write access times; caching prevents the system from having to perform more input and outputs from a slow hard drive. Perhaps, we can decouple encryption and decryption to remove the effect of caching on our times.

Finally, the asymmetric ciphers of DSA and ECC are comparable to hashes in terms of running times. We believe this is because they first hash the message before performing any computationally intensive operations. In contrast, RSA performs operations over the entire message, since it is not just a signing and authentication cipher. We believe that we have some errors in our RSA encryption data; RSA encryption took a very long time, so we were only able to take a limited number of time measurements. Some of these measurements may have been skewed, since a long-running job can be give lower and lower priority as it runs longer and longer. Also, since it runs for so long it is more susceptible to other variables such as fluctuating server load.

RSA encryption we believe to be modeled by a polynomial of degree two. We believe that RSA decryption is linear. Theoretically, however we believe that decryption should grow at the same rate or faster than encryption. Perhaps, we need to take more measurements. This was difficult to do since RSA was extremely slow. Measurements would need to be taken at smaller file size and with smaller keys to get a better idea. By our values of R^2, we tentatively conclude that RSA encryption grows polynomially and decryption grows linearly.

REFERENCES

AOL. (2016). Camellia (Cipher). Retrieved November 29, 2016, from http://www.wow.com/wiki/Camellia\_(cipher)

Barreto, P. (November 25, 2008). The Whirlpool Hash Function. Retrieved November 20, 2016, from http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html

Bosselaers, A. (2012). The Hash Function RIPEMD-160. Retrieved November 18, 2016, from http://homes.esat.kuleuven.be/~bosselae/ripemd160.html#Speed

Crypto, M. (2004). SEED. Wikipedia. Retrieved November 22, 2016, from https://en.wikipedia.org/wiki/SEED

Free Software Foundation. (2003). Class Cast5. Retrieved November 24, 2016, from https://www.gnu.org/software/gnu-crypto/manual/api/gnu/crypto/cipher/Cast5.html

Frost, J. (1970). Regression Analysis: How to Interpret S, the Standard Error of the Regression | Minitab. Retrieved December 04, 2016, from http://blog.minitab.com/blog/adventures-in-statistics/regression-analysis-how-to-interpret-s-the-standard-error-of-the-regression

Kelsey, J., & Schneier, B., & Wagner, D. (n.d.). Related-Key Cryptanalysis of 3-WAY, Biham-DES, CAST, DES-X, NewDES, RC2, and TEA. Retrieved November 22, 2016, from ttps://www.schneier.com/academic/paperfiles/paper-relatedkey.pdf

RIPEMD-160. (2016). Retrieved November 16, 2016, from http://rosettacode.org/wiki/RIPEMD-160

RIPEMD-160. (2015). Retrieved November 29, 2016, from https://en.bitcoin.it/wiki/RIPEMD-160

Roberts, D. (n.d). Statistics 2 – Correlation Coefficient and Coefficient of Determination. Retrieved November 27, 2016, from http://mathbits.com/MathBits/TISection/Statistics2/correlation.htm

Stallings, W. (2006). The Whirlpool Secure Hash Function. *Cryptologia,* 30(1), 55-67. Doi: 10.1080/016111905003800890

Vocal Technologies. (2015). Camellia. Retrieved November 20, 2016, from https://www.vocal.com/cryptography/camellia/

Vocal Technologies. (2015). RIPEMD. Retrieved November 24, 2016, from https://www.vocal.com/cryptography/ripemd/

World Library. (1990-1993). The Electronic Version of The Complete Works of William Shakespeare. Retrieved November 27, 2016, from https://ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/files/t8.shakespeare.txt

Yu, T. (2012). Projects/Camellia Encryption. Retrieved November 30, 2016, from http://k5wiki.kerberos.org/wiki/Projects/Camellia\_encryption

Appendix A

Appendix B

Appendix C