Investigating the Speed and Security of Cryptographic Key Based Hashing Algorithms

How effective in terms of speed and security is the AES-CBC-256 encryption algorithm when compared to the RSA-4096 encryption algorithm?

A Computer Science Extended Essay

Word Count: 3945

Finn Lestrange

August 2021

Contents

Abstract

The following research paper investigates the speed and security of the two most popular symmetric and asymmetric encryption algorithms, AES and RSA respectively. The paper provides some background information and explains the mathematics required to understand how these encryption algorithms work, discussing the AES Cipher Block Chaining Mode and RSA 4096. The algorithm testing carried out was done using the Oracle Java programming language and for each encryption algorithm there was a separate class that contained a key generation, encryption and decryption methods making use of the Java system time method to count how long each of these cryptographic operations for each of the separate algorithms took. These programs were run on different sizes of pseudo random integer and character input data and the results were collected for key generation and both the encryption and decryption speed per set of input data. The major findings of this paper are as follows; AES is a more versatile encryption algorithm with a much wider range of use cases due to the rapid encryption and decryption times provided by the robust symmetrical algorithm. Whereas RSA is better suited to encrypting small amounts of data for digital communications, due to the very secure keys but huge key generation times and un-crackable ciphertext. The most notable data collected is as follows; key generation times, AES $\approx$ 0.205 ms, RSA $\approx$ 1386 ms, resulting in a 6762x increase in key generation time from AES to RSA. Thus, highlighting why RSA is better suited to non-time critical messaging and why AES is preferred for fast-access digital encryption, e.g. web technologies and confidential document storage.

1. Introduction

Cryptographic hashing algorithms are computer programs or pieces of code that can create a key that encrypts data such that the key has to be used to decrypt and read the data again. They can be found in almost all applications of digital technology, from mobile phones, to ATMs, email providers etc. as they are intended to allow data to be stored in such a way that only valid users are meant to be able to have access.

There are a large number of hashing algorithms out there ranging from extremely complicated and highly secure, to fast and efficient algorithms designed with speed over security, thus the choice of algorithm can be difficult when designing a product or solution that requires data to be encrypted.

This paper will investigate two cryptographic hashing algorithms RSA-4096 & AES-CBC-256 and how different sample sizes of pseudo-random data affects both the speed of encryption and the speed of decryption. This paper will also provide explanations of the mathematics behind how these hashing algorithms work and will derive some typical and reasonable use cases for each of the algorithms based on their performance in the tests.

This research could be useful to programmers who are trying to decide the right hashing algorithm when designing a program, and for security researchers or companies that want to compare which algorithm is right for their intended use case. Hashing algorithms are used all the time and thus the vast number of them can make it confusing as to which ones may be best suited to the largest number of general tasks, thus this is the reason why I have chosen both a symmetric and an asymmetric algorithm to compare.

To compare these algorithms, two programs will be written and run in real time, one that preforms AES encryption and decryption and another that performs RSA encryption and decryption and they will be tested on different sizes of pseudo random integers and ascii characters. The time for encryption and decryption will be recorded for each run using Java's built in System timer, `System.currentTimeMillis()`, and then the times for the increasing sizes of data for both operations can be stored and analysed. This approach is also heavily dependant on the users hardware and as such the results obtained here may not accurately reflect results that may be obtained using the same methods.

2. Background Information

2.1 256-bit Hashing Algorithms vs. 4096 bit Key Based Algorithms

A hashing algorithm is "mathematical algorithm (or function) that maps data of arbitrary size to a hash of a fixed size" [1]. In essense, all hashing algorithms take some input data and perform a mathematical operation on the bytes of the data such that the output of the algorithm can only be read using the key that was used to encrypt the data. This key based system is the backbone for both the AES (Advanced Encryption Standard) hashing algorithm and the RSA (Rivest, Shamir, Adleman) hashing algorithm. Both algorithms utilise key based systems used for encryption and decryption of data. However the distinct difference between the two algorithms is that AES uses a single 256-bit key for both encryption and decryption, called symmetric encryption, whereas RSA uses a two key system, one for encryption, called the public key and one for decryption called the private key, aptly named asymmetric encryption. [2] The size of the keys used are also vastly different, as AES can use 128, 192 or 256 bit keys and RSA can use 512, 1024, 2048, 3072 and 4096 bit keys.

Although they are both key based systems, it makes a good comparison as both of these encryption algorithms essentially do the same thing; encrypt data using keys, but these algorithms take very different approaches in terms of their cryptography behind manipulating data to be encrypted or decrypted.

2.2 AES

The Advanced Encryption Standard is "a data encryption standard endorsed by the U.S. National Institute of Standards and Technology (NIST) as a replacement for aging and weak Data Encryption Standard (DES); adopted by the NIST in the late 1970s. The Advanced Encryption Standard offers far greater security than DES for communications and commercial transactions over the Internet."[3]. AES was invented by two Belgian computer scientists, Joan Daemen and Vincent Rijmen and was introduced in the year 2000 after the NIST put out a request for people to create a new encryption standard. AES was selected by the NIST and has since been accepted as the new standard for data encryption for the United States Government. This is due to the flexibility of the algorithm having varying key lengths and that the mathematical algorithm behind AES is able to utilise newer, more modern computer architecture. Additionally, AES is less computationally intensive than its predecessor DES, thus making it both a sensible and suitable choice as the new encryption standard [3].

As AES was the new standard for encrypting digital communications, naturally it was picked apart by the crypto-security community and the initial implementation of AES encryption, called ECB mode or Electronic Code Book had a few flaws. Mainly that the cipher did not incorporate any randomness after the key had been generated, thus if the same key was used on the same input data, it would always produce the same output, also known as the ciphertext. This meant that more modes of AES encryption were developed and the one that was adopted the fastest was a mode of encryption called Cipher Block Chaining Mode or CBC for short. This mode made use of a cryptographic technology that had been patented in 1977 by four engineers, William F. Ehrsam, Carl H. W. Meyer, John L. Smith and Walter L. Tuchman [4]. This cipher mode used "successive cycles of operation during each of which an input block of clear data bits is ciphered under control of an input set of cipher key bits to generate an output block of ciphered data bits" [4]. This cipher also made use of a chaining mode; hence the named Cipher Block Chaining, as for each new piece of data to be encrypted is uses part of the previous data that was encrypted, thus individual blocks of data cannot be decrypted, which was one of the main weaknesses with the initial AES implementation. Due to it's robust nature, the AES-CBC algorithm has been adopted as the go-to AES method for fast encryption and decryption, and it is the algorithm that will be used throughout the rest of this paper.

2.2.1 Typical Use Cases

Although the AES encryption standard was created for use by the U.S National Institute of Standards and Technology (NIST) it is now one of the most widely used cryptographic standard for the encryption and decryption of data. AES and it's subsidiary encryption modes (CBC, CFB etc.) are used in internet communications and data transmission due to its speed and relative security [2]. As AES is a symmetric encryption cipher, it is much better at storing large amounts of data for long periods of time, this is one of the main reasons why it has become so popular and why the NIST was so quick to adopt it as their preferred method for securing digital information.

2.2.2 Mathematical Algorithm

The mathematics explained in this section will be relating to the Cipher Block Chaining implementation of the AES cipher. Something else to note is that due to the nature of AES and how it is built upon so many other pieces of cryptography, this will have to be a very simplified version of the mathematics behind how it works, both to keep it relevant and to ensure that the rest of this paper can be understood without confusion.

Overall, AES uses a fairly robust algorithm that is based on many other algorithms, and as such it is more of a culmination of many mathematical and cryptographic functions that all work together to make AES work. This is where AES is very unlike RSA, as RSA relies mostly on mathematical concepts. However, like the RSA algorithm that will be explained in this paper later on, AES uses padding to ensure that the data being encrypted is all the same size, this means that encrypted data is usually larger than the original, un-encrypted data.

Key & IV Generation

Both the key and IV generation processes are quite simple. As the same key is used for both the encryption a decryption operations, the can can be any secure random number that is of the specified key length, typically 128 or 256 bits in length. The Initialization Vector (IV) used in AES is what is called a cryptographic nonce, meaning that it is a securely random number that is only used in one cryptographic operation. The IV is generated by filling a 12 byte array with a cryptographically secure random number or CRNG for short. The key is generated in much the same way, an empty byte array is initialized of the specified key size, then it is filled using the CRNG.

Encryption

Once the key and IV have been generated they can be used to encrypt data. This process has six steps:

1. Key Expansion
2. Data Mixing
3. Substitution

4. Shifting of Rows
5. Shifting of Columns
6. Repetition of Rounds

AES encryption is done using a process called rounds, and depending on the key size, the number of rounds varies, for example, a 128 bit key means that there will be 10 rounds and a 256 bit key uses 14 rounds [5].

The first step to take place is the key expansion, where by the AES key is used to create a new set of cryptographic keys called round keys, with a new one being generated for each round. This is done using the original AES key and a very complex algorithm called Rijndael's key schedule algorithm [5]. This algorithm takes the AES key and performs various mathematical operations on it to produce smaller keys that can be used for each round of encryption.

After the round key has been generated, the data that we want to encrypt is used in the XOR bitwise operation with the current round key and the Initialization Vector, not shown in diagram. This combines the round key and the plaintext data where it is then stored in a matrix. The following diagram illustrates this process, where the $\oplus$ signifies the XOR operation.

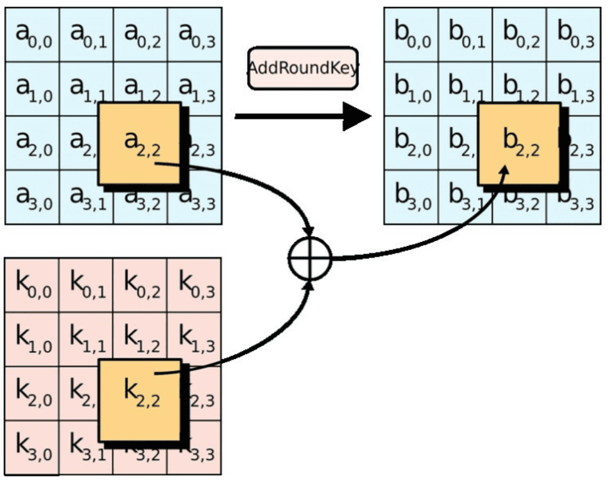


Figure 2.1: Bitwise operation with addition of round key to plaintext input, [5]

The data is then used in an algorithm called a substitution permutation network, that uses substitution bytes and a very complex mathematical algorithm to essentially mix up the data in a specified way. Again, below is a very simplified diagram of the process.

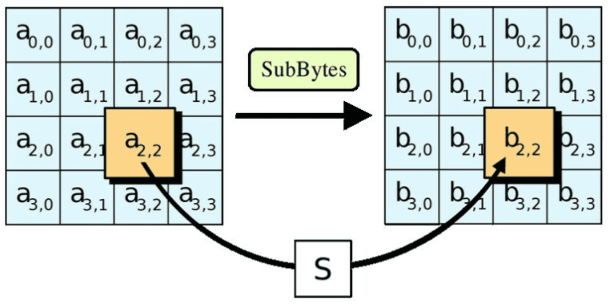


Figure 2.2: Substitution permutation network, [5]

As the data is arranged in a matrix, we can treat it almost like you would with a spreadsheet, by manipulating the rows and columns. This starts by shifting every 8 bits that makes up the 128 or 256 block to the right by one place.

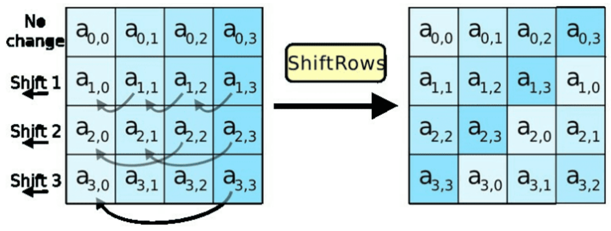


Figure 2.3: Shifting rows 1 byte to the right, [5]

Finally, the columns of data in the matrix can be shuffled, illustrated in the diagram below, where c (x) denotes the algorithm that shuffles the columns.

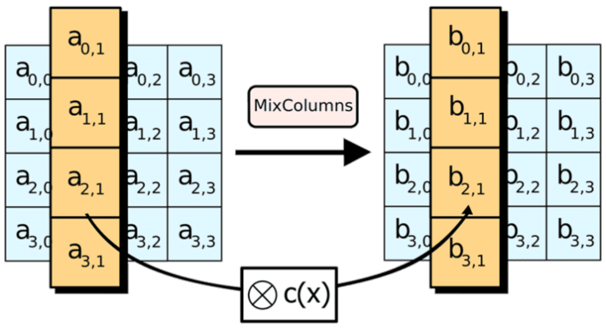


Figure 2.4: Mixing the columns, [5]

This process is then repeated for the number of rounds specified by the key size and the output data from this is the ciphertext and is the encrypted version of the input data we gave it.

Decryption

The decryption process is the reverse of all of the steps above, however one crucial detail is that you need to find the inverse round keys and to do that you must have the original AES key that was used to generate the original round keys.

2.3 RSA

Ths Rivest, Shamir & Adleman Encryption Algorithm or RSA for short is a asymmetrical (public and private key based) cryptographic hashing algorithm developed by staff working at Massachusetts Institute of Technology, Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. The very large key sizes utilized by the algorithm and the complex mathematical structure behind it make it both useful for both people and government agencies like the US National Security Agency who adopted RSA for their communications [6]. The RSA algorithm was released in 1977 and is now the most widely used asymmetrical encryption algorithm, as it allows information to be exchanged using a publicly known secret for users to encrypt data with and a private secret that the receiver of information uses to decrypt the data [7].

In essence the RSA asymmetric encryption algorithm works by generating really large numbers and then it checks if they are prime, when it finds the two large prime numbers they are then used to create both the public and private keys [8]. However, there are many more steps in this process that will be explained in further detail in *Section 2.3.2*.

2.3.1 Typical Use Cases

RSA encryption can be found on internet services such as Virtual Private Network (VPN) providers networks, secure email services; like ProtonMail, and secure messaging apps like Signal. RSA encryption is slowly making it's way into out digital lives even more are companies are realizing that it is a very effective way of keeping user communications and data secure. By design the companies that hold user data encrypted with RSA rely on the users remembering or keeping their private key. Thus the only people that can read data encrypted using RSA are the people who know both the public and the private key.

2.3.2 Mathematical Algorithm

The current implementation of RSA is heavily mathematical in nature, using modular arithmetic and mathematics of Euler, including The Totient Function, $\phi(n)$ [9]. The following explanation of the mathematics behind RSA will be split up into 3 sections; *Key Generation*, *Encryption of Plaintext* and *Decryption of Ciphertext.*

Key Generation

As mentioned in *Section 2.1*, RSA is an asymmetrical encryption algorithm, thus it makes use of two keys, one for data encryption and another for the decryption of data. These keys are called the public and private keys respectively. The key generation process is centered around the generation of $1024 \longrightarrow 4096$ bit numbers such that the product of two prime numbers is equal to that large number.

Public Key Generation

The first step in key generation is for the user, with the aid of a computer program to choose two large prime numbers that meet the length requirements of the key that they want to generate. For this explanation we will be using p % q to denote these primes.

Next, the user calculates the product of these prime numbers such that the prime factors of this new number n and the two primes p & q are the prime factors of this number.

$$n = p \cdot q \tag{2.1}$$

This product n is then half of the public encryption key.

Next, the user calculates a number e, making use of Euler's Totient Function $\phi\left(x\right)$ such that e is relatively prime to n.

$$\phi\left(p \cdot q\right) = \left(p - 1\right)\left(q - 1\right) \tag{2.2}$$

This number e is then the other half of the public encryption key. Thus the private key is the combination of $\left(n,\ e\right)$

Private Key Generation

The user now calculates d, the modular inverse of e.

$$d = e \cdot \mathrm{mod}\ \phi(n) \tag{2.3}$$

Such that,

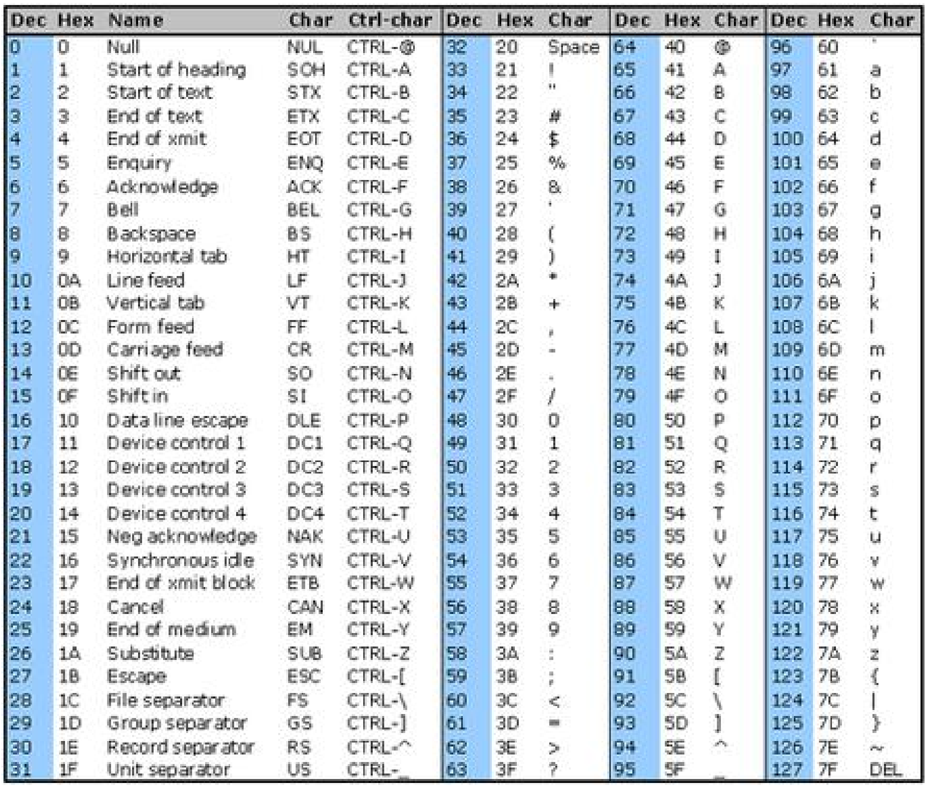
$$d \cdot e = 1 \cdot (\mathrm{mod}\ \phi(n)) \tag{2.4}$$

This means that d is the private key. Now the public key can be sent out for users to encrypt data and the receiver then uses d to decrypt the data [9].

Encryption of Plaintext

Once the private key has been sent out, another user can then encrypt a message using it. The following steps are used to perform this operation.

First a user converts their message into a number m, using the ASCII alphabet, using either computer software or a chart like the one below [9].



Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(	72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29	)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-_	63	3F	?	95	5F	_	127	7F	DEL

Figure 2.5: ASCII Chart [10]

The users message, in ASCII form, is then raised to the power e (the second half of the public key), and then multiplied by the modulo of the first half of the public key, n.

$$c = m^e \cdot (\text{mod } n) \tag{2.5}$$

Where c is the ciphertext that is then sent to the person who has the private key to decrypt the message.

Decryption of Ciphertext

Once the owner of the private key has received the ciphertext c, they use Euler's Theorem to retrieve the original message in ASCII form. As Euler's Theorem states:

$$m^{\phi(n)} \equiv 1 \cdot \text{mod}\,(n) \tag{2.6}$$

we need the integer d that is the private key because d satisfies the relationship:

$$d \cdot e \equiv 1 \cdot (\bmod\ \phi(n)) \tag{2.7}$$

Thus with the knowledge of d the user can apply the mathematical relationships in *Equations 2.6 & 2.7* to discover the message integer m. Which can finally be converted from ascii into the original message from the sender. [9]

3. Experiment Methodology

The process of experimentation for testing the speed of these two algorithms is quite simple, there will be 10 trials where each algorithm will have to encrypt pseudo-random ASCII characters. As each trial goes on, 50 more characters will be added to the plaintext input data. The first trial will start with 50 characters and each set of characters will be the same for each algorithm. The time taken to generate the key, encrypt the data and the time taken to decrypt the data will be stored in a csv file to later be used for data analysis.

Please Note: *The results derived from this experiment may vary from results obtained using the same or a similar process on your own computer, this is due to the specifications of said device as this has an effect on the speed that data can be encrypted and decrypted.*

3.1 Dependant Variables

3.1.1 Time

To gather data for this experiment, the time taken for the key generation, and encryption / decryption of various sizes of pseudo random character and integer data sets will be recorded and used in the data analysis *Section 4.3.*

The time will be recorded using the Java built in system timer, `System.currentTimeMillis()` and the results will be saved to a csv file.

3.2 Controlled Variables

- Pseudo-Random Data Sets $\longrightarrow$ As this experiment will involve the comparison of two encryption algorithms, we must ensure that the data being used is the same between the two algorithms.
- Computational Power $\longrightarrow$ As encryption and decryption speeds vary based on computer hardware, all of the tests will be run on the exact same desktop workstation using an Intel® Core i7 5930K.

3.3 Experimental Procedure (AES-CBC-256)

To test the performance of the AES-CBC-256 encryption algorithm, a class was written that contained the following methods:

- ivGen $\longrightarrow$ used to generate an Initialization Vector to be used for the encryption and decryption of plaintext
- keyGen $\longrightarrow$ used to generate an AES-256 bit key
- encrypt $\longrightarrow$ that takes a parameter of `String plaintext` to be encrypted
- decrypt $\longrightarrow$ that takes a parameter of `String ciphertext` to be decrypted

3.4 Experimental Procedure (RSA-4096)

To test the RSA-4096 cryptographic algorithm, a class was also written containing the following methods:

- keyGen $\longrightarrow$ used to generate an RSA-2048 `keyPair Object`
- encrypt $\longrightarrow$ that takes a parameter of `String plaintext` to be encrypted
- decrypt $\longrightarrow$ that takes a parameter of `String ciphertext` to be decrypted

Finally, there is a parent class that has access to the methods from both the AES and RSA classes and that automates the testing and the data collection.

4. Experiment Results

4.1 Tabular Data Presentation

Listed below are tables of the averages for the key generation, encryption and decryption times in milliseconds for both the AES and RSA algorithms.

AES - Key Generation Times										
	Input Data Size (Characters)									
	50	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
Average Time (Milliseconds)	2	0	0	0	0	0	0	0	0.05	0

Figure 4.1: AES Key Generation Times (Averages)

AES - Encryption Times										
	Input Data Size (Characters)									
	50	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
Average Time (Milliseconds)	0.25	0.05	0	0.05	0.2	0.15	0.1	0.1	0	0.15

Figure 4.2: AES Encryption Times (Averages)

AES - Decryption Times										
	Input Data Size (Characters)									
	50	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
Average Time (Milliseconds)	0.1	0.05	0.15	0.15	0.1	0.05	0.1	0.15	0.05	0.1

Figure 4.3: AES Decryption Times (Averages)

RSA - Key Generation Times					
	Input Data Size (Characters)				
	50	**100**	**150**	**200**	**250**
Average Time (Milliseconds)	1392	1532.25	1509.85	1422.2	1069.65

RSA - Key Generation Times					
	Input Data Size (Characters)				
	300	**350**	**400**	**450**	**500**
Average Time (Milliseconds)	1539.35	1433.65	1074.7	1223.6	1664.7

Figure 4.4: RSA Key Generation Times (Average)

RSA - Encryption Times										
	Input Data Size (Characters)									
	50	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
Average Time (Milliseconds)	0.25	0.4	0.2	0.35	0.3	0.45	0.2	0.5	0.3	0.2

Figure 4.5: RSA Encryption Times (Averages)

RSA - Decryption Times										
	Input Data Size (Characters)									
	50	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
Average Time (Milliseconds)	10.05	11.25	11.05	9.85	10.2	10.3	9.5	10.15	11.2	9.65

Figure 4.6: RSA Decryption Times (Averages)

4.2 Graphical Data Presentation

Please Note: *The following graphs are for the individual results as aggregating the results for each category for each algorithm into one graph would not be possible due to the orders of magnitude difference for some of the results.*

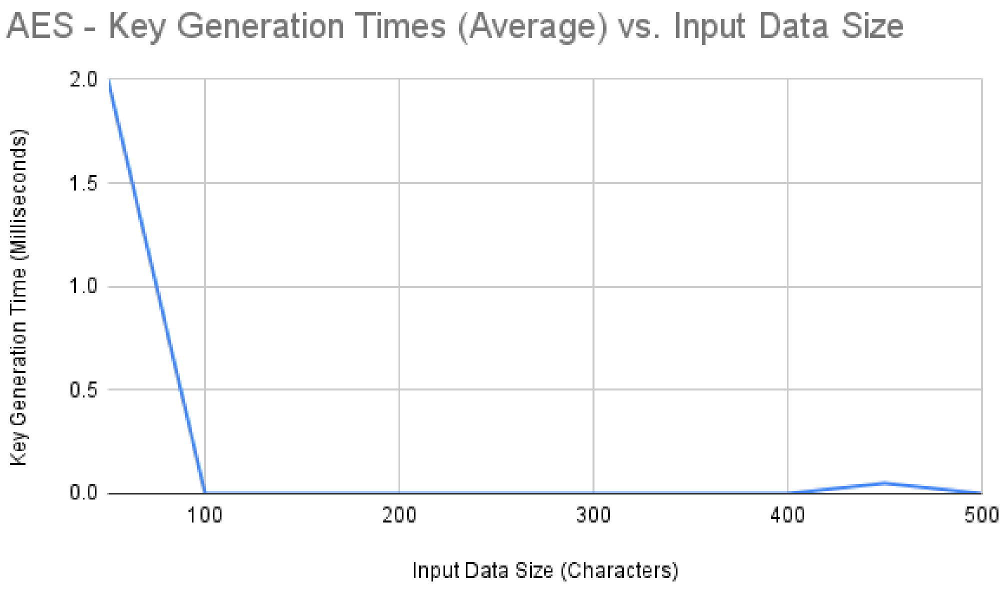


Figure 4.7: AES Key Generation Time vs. Input Data Size

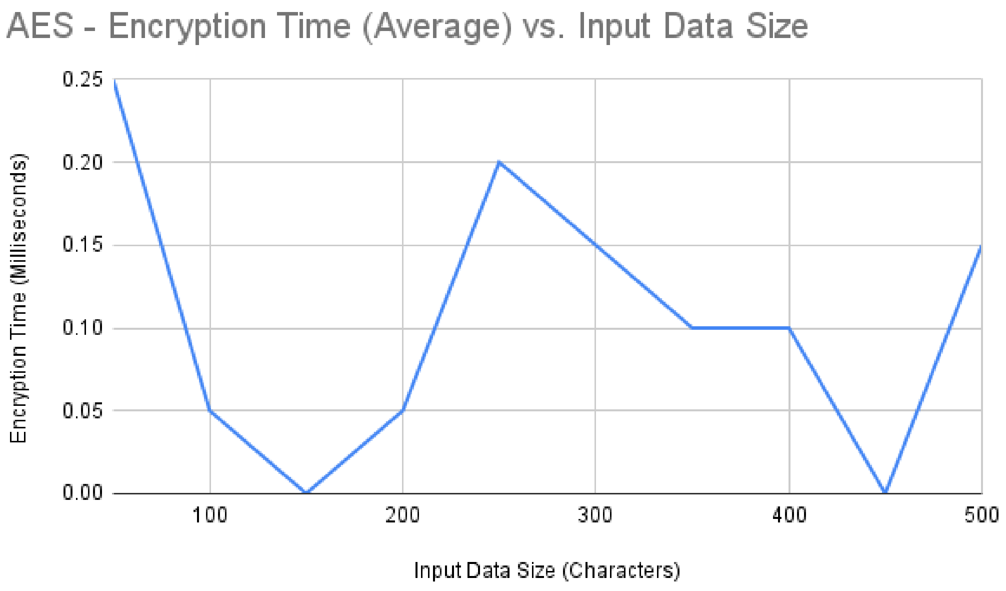


Figure 4.8: AES Encryption Time vs. Input Data Size

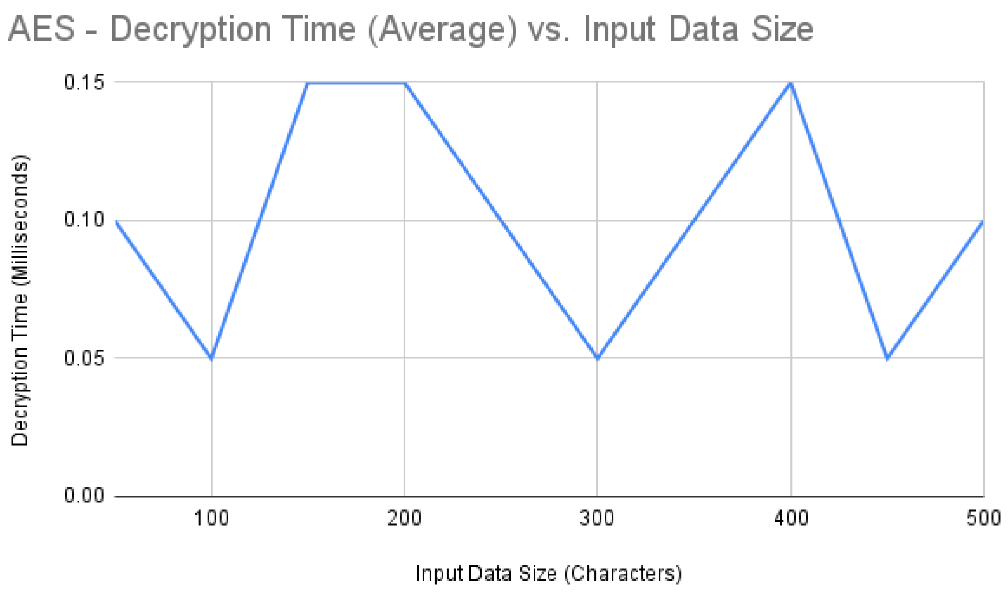


Figure 4.9: AES Decryption Time vs. Input Data Size

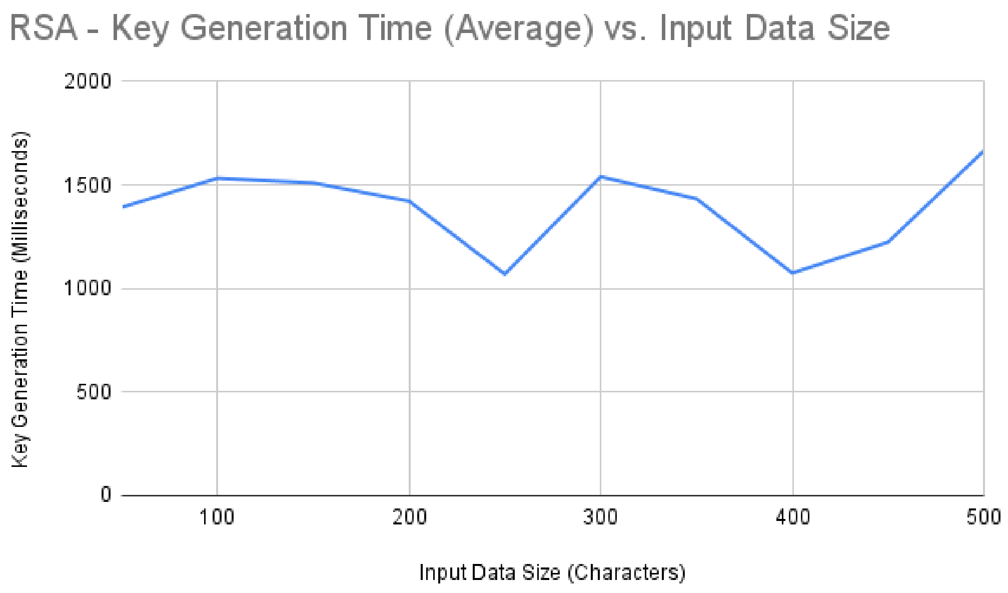


Figure 4.10: RSA Key Generation Time vs. Input Data Size

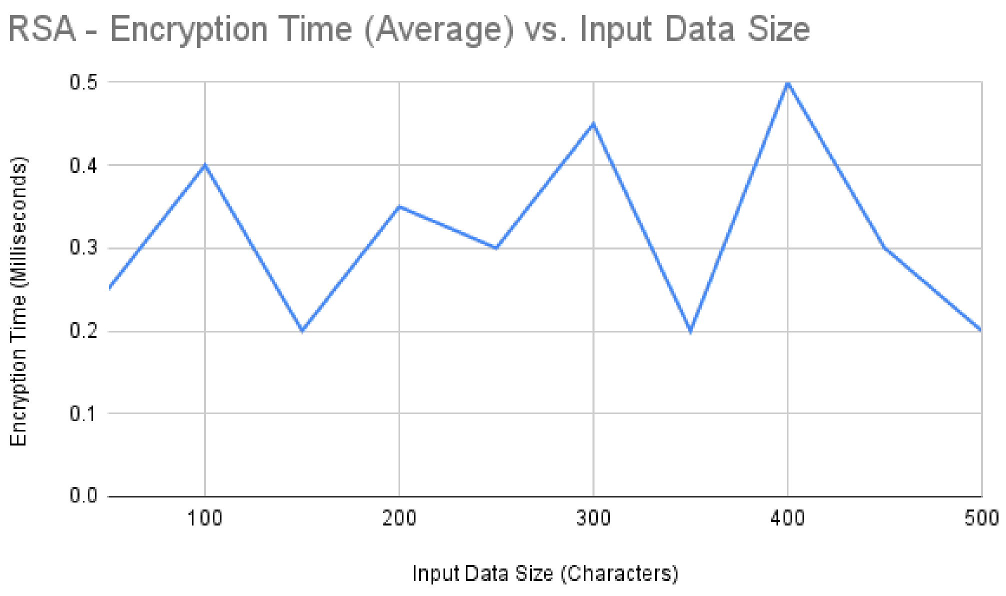


Figure 4.11: RSA Encryption Time vs. Input Data Size

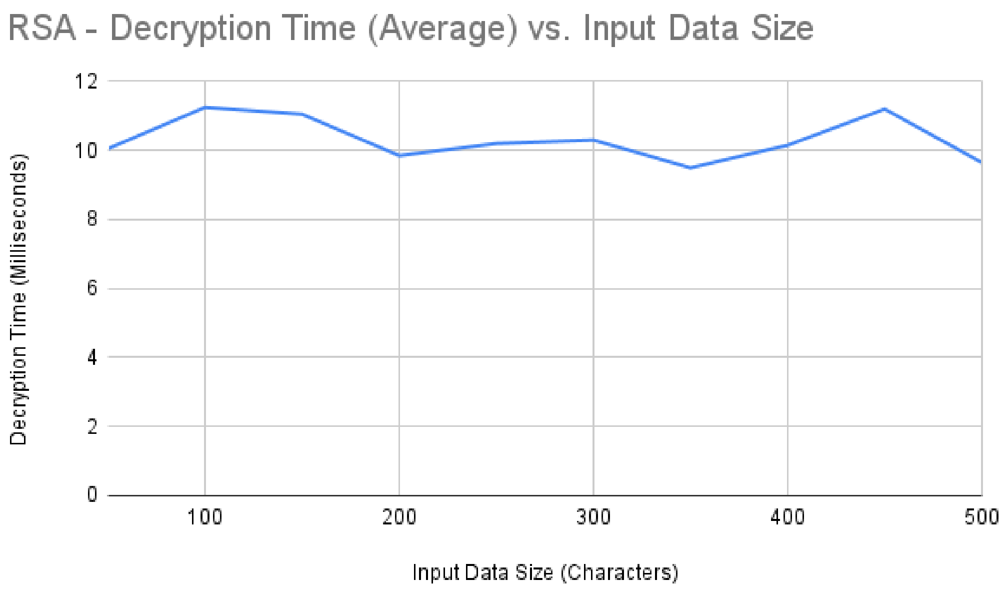


Figure 4.12: RSA Decryption Time vs. Input Data Size

4.3 Data Analysis

The following section will explain the trends and patterns in the data. We will analyze and compare the key generation times, encryption times and decryption times for the results gathered from the AES and RSA data.

4.3.1 Analyzing Key Generation Times

From tables *4.1* and *4.4* it can be observed that clearly the key generation process for RSA-4096 is exponentially higher than AES-256, with the average key generation time being less than a millisecond whereas if we average the key generation times for RSA, that comes out at:

$$\text{AES KeyGen Average} = 0.205 \text{ ms} \tag{4.1}$$

$$\text{RSA KeyGen Average} = 1386.195 \approx 1386 \text{ ms} \tag{4.2}$$

This also means that the average RSA key generation time takes ≈ 6762 times longer to generate a set of keys than AES takes to generate it's respective key.

$$\frac{1386.195}{0.205} \approx 6762 \tag{4.3}$$

This is the first of many indications as to the ideal use case for RSA.

Another observation to note is that there are a few anomalous results, namely the average key generation time for AES when generating a key for the 50 & 450 character data sets. This small rise in the time taken could have been caused by a lack of system entropy or another process on the computer using the systems secure entropy source. This in terms of AES doesn't have a significant effect on the time as a maximum key generation time of 2 ms is almost a negligible difference in the grand scheme of things.

Something else of note is that for the AES key generation times, the first run usually takes longer to generate, this could be caused again by the systems secure entropy source being used by another process or that it needs to be initialized.

In terms of RSA, the key generation times seem entirely random, and they are also very high when compared to AES. However, this fits with what we know about the RSA Mathematical Algorithm and how the RSA keys are generated as the process is very computationally intensive. Thus far the data collected supports what we know about the two algorithms.

4.3.2 Analyzing Encryption Times

When we observe the data for the encryption times, the times for the two algorithms are at least in the same order of magnitude, unlike the times for key generation, with all of the encryption times under 1 ms. For both cryptographic algorithms the times for data encryption seem to be entirely unrelated to the size of the input data set, this can be observed in *Graphs 4.8 and 4.11.* As we can only speculate about why this happens, I would suggest that these seemingly uncorrelated results could be due to the algorithms having to add padding to the plaintext input data.

4.3.3 Analyzing Decryption Times

When observing that data for decryption times, it can be noted that the decryption times for the AES algorithm is a single order of magnitude smaller than the decryption times for RSA, with the following averages being calculated:

$$\text{AES Decryption Time Average} = 0.1 \text{ ms} \tag{4.4}$$

$$\text{RSA Decryption Time Average} = 10.32 \text{ ms} \tag{4.5}$$

Here we see another case where the RSA algorithm is performing as we would expect as the decryption process for RSA is more computationally intensive when compared to the process the AES uses to decrypt data.

5. Conclusions

When drawing conclusions from this data and from what has been discussed about how these two cryptographic algorithms work, something that needs to be kept in mind is that AES is a symmetric encryption algorithm and RSA is an asymmetric algorithm. This means that by design RSA is meant to be more secure and the fact that is had to generate two keys that are 16 times larger than the single key that AES has to generate. This also means that the data collected fits exactly with what is known about the RSA algorithm and as such it can be said with confidence that the data collected is valid and accurate as the testing was completely automated and and had no human interaction. The code was also written in a robust and versatile programming language, Java in this case, and this helps to eliminate that any issues with the language that could have affected the outcome of the tests.

Something of note from both the research and the experiment is that the choice of using RSA to compare to AES may not have been the best comparison in terms of determining which of the two are better. This is because the two algorithms are entirely different in their approach and thus it makes them challenging to compare. However, the results of this investigation do demonstrate to us that both of the algorithms do have use cases that they are best suited to.

From the data analyzed we can see that the RSA algorithm is best suited to sending digital communications and small messages, that are not time and space critical due to the high time and space complexity. The data also shows us that the AES algorithm is much more versatile and is best suited for storing large amounts of data as the time for key generation, decryption and encryption is small in comparison to the RSA algorithm.

This research paper will hopefully prove useful to developers in helping to guide them in choosing a symmetric or asymmetric encryption algorithm for the storage and transmission of data, this ultimately will help to make future applications and our digital information more secure.

References

[1] J. Code, *Hashing algorithms — jscrambler blog*, Jscrambler. [Online]. Available: https://blog.jscrambler.com/hashing-algorithms (visited on 07/26/2021).

[2] A. M. Abdullah, *Advanced encryption standard (aes) algorithm to encrypt and decrypt data*, ResearchGate, Jun. 2017. [Online]. Available: https://www.researchgate.net/publication/317615794_Advanced_Encryption_Standard_AES_Algorithm_to_Encrypt_and_Decrypt_Data (visited on 08/07/2021).

[3] G. J. Simmons, *AES — cryptology — Britannica*. 2020. [Online]. Available: https://www.britannica.com/topic/AES.

[4] W. F. Ehrsam, C. H. W. Meyer, J. L. Smith, and W. L. Tuchman, *Message verification and transmission error detection by block chaining*, Google Patents, 1977. [Online]. Available: https://patents.google.com/patent/US4074066A/en (visited on 08/22/2021).

[5] D. Crawford, *A complete guide to aes encryption (128-bit and 256-bit) - proprivacy.com*, ProPrivacy.com, Feb. 2019. [Online]. Available: https://proprivacy.com/guides/aes-encryption.

[6] G. J. Simmons, *RSA encryption — Britannica*. 2020. [Online]. Available: https://www.britannica.com/topic/RSA-encryption.

[7] *Rsa cryptography: History and uses – telsy*, Telsy, May 2021. [Online]. Available: https://www.telsy.com/rsa-cryptography-history-and-uses/ (visited on 08/16/2021).

[8] P. Fox, *Public key encryption (article)*, Khan Academy. [Online]. Available: https://www.khanacademy.org/computing/computers-and-internet/

xcae6f4a7ff015e7d:online-data-security/xcae6f4a7ff015e7d:data-encryption-techniques/a/public-key-encryption.

[9] A. Katz, A. Ng, and P. Bourg, *Rsa encryption — brilliant math and science wiki*, Brilliant.org, 2010. [Online]. Available: https://brilliant.org/wiki/rsa-encryption/.

[10] commfront, *Ascii chart*, CommFront. [Online]. Available: https://www.commfront.com/pages/ascii-chart (visited on 08/22/2021).

6. Appendix

6.1 Source Code

Below is the Java 16 source code that was used with the experiment with the experimental procedures to collect the raw data.

```java
// Finn Lestrange - 27/08 - Extended Essay Code -> Main Class File

// Imports
import javax.crypto.*;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import java.io.*;
import java.nio.charset.StandardCharsets;
import java.security.*;
import java.util.Base64;
import java.util.LinkedList;
import java.util.Locale;
import java.util.Random;

public class Algorithms {

    // LinkedLists to store the times taken for the various stages of
       the algorithms
    protected static LinkedList<Long> encTimeAES = new LinkedList<>();
    protected static LinkedList<Long> decTimeAES = new LinkedList<>();
    protected static LinkedList<Long> keyTimeAES = new LinkedList<>();
    protected static LinkedList<Boolean> validAES = new LinkedList<>();
    protected static LinkedList<Long> encTimeRSA = new LinkedList<>();
    protected static LinkedList<Long> decTimeRSA = new LinkedList<>();
    protected static LinkedList<Long> keyTimeRSA = new LinkedList<>();
    protected static LinkedList<Boolean> validRSA = new LinkedList<>();

    // LinkedList to store the randomly generated strings
    protected static LinkedList<String> randomStrings = new LinkedList
       <>();



    // RSA Encryption and decryption class
    static class RSA {

       public static KeyPair keypair;

       public static void keyGen() {
           try {
               // Creates a new instance of the keyGenerator class for
                   RSA
               KeyPairGenerator keyPairGenerator = KeyPairGenerator.
                   getInstance("RSA");

               // Initializes the keyGenerator with an RSA key of size
                   2048
               keyPairGenerator.initialize(4096);

               // Stores the keyPair in the global variable so that it
                   can be used by the other methods
               keypair = keyPairGenerator.generateKeyPair();
           } catch (NoSuchAlgorithmException e) {
               e.printStackTrace();
           }
       }

       public static String encrypt(String plaintext) {
           String b64out = "";
           try {
               // Initializes a new instance of the RSA cipher in
                   encrypt mode using the public key generated before
               Cipher cipher = Cipher.getInstance("RSA");
               cipher.init(Cipher.ENCRYPT_MODE, keypair.getPublic());

               // Gets the UTF_8 bytes of the plaintext to be
                   encrypted
               byte[] plaintextBytes = plaintext.getBytes(
                   StandardCharsets.UTF_8);

               // Generated the ciphertext from the plaintext
               byte[] ciphertext = cipher.doFinal(plaintextBytes);

               // Converts the encrypted bytes to a base64 string for
                   ease of use and storage
               b64out = Base64.getEncoder().encodeToString(ciphertext)
                   ;
           } catch (NoSuchPaddingException | NoSuchAlgorithmException
               | InvalidKeyException | IllegalBlockSizeException |
               BadPaddingException e) {
               e.printStackTrace();
           }
           return b64out; // returns the base64 encoded ciphertext
       }

       public static String decrypt(String ciphertext) {
           String message = "";
           try {
               // Initializes the cipher class for RSA decrypt mode
                   using the private key generated before
               Cipher cipher = Cipher.getInstance("RSA");
               cipher.init(Cipher.DECRYPT_MODE, keypair.getPrivate());
               // Decodes the base64 ciphertext into a byte array
               byte[] cipherTextNoB64 = Base64.getDecoder().decode(
                   ciphertext);

               // Decrypts the ciphertext into a byte array
               byte[] messageBytes = cipher.doFinal(cipherTextNoB64);

               // Converts the decrypted byte array into a human
                   readable String
               message = new String(messageBytes);
           } catch (NoSuchPaddingException | IllegalBlockSizeException
                | NoSuchAlgorithmException | BadPaddingException |
               InvalidKeyException e) {
               e.printStackTrace();
           }
           return message; // returns the decrypted ciphertext
       }
   }

   // AES Encryption and decryption class
   static class AES {

       public static IvParameterSpec ivSpecG;
       public static SecretKeySpec keySpecG;



       // AES Key generating method
       public static void keyGen() throws NoSuchAlgorithmException {

           // Create a new key generator for AES
           KeyGenerator keyGenerator = KeyGenerator.getInstance("AES")
               ;
           keyGenerator.init(256); // Initializes the keyGenerator
               with a 256 bit key
           SecretKey sKey = keyGenerator.generateKey(); // Generates
               the key

           // Creates a new secretkey for the AES algorithm
           // This method converts a generic 256 bit key to a 256 bit
               aes key
           // saves the key to the global variable so that it can be
               used in the decryption method without
           // having to worry about saving to files and reading from
               files -> adds unnecessary complications
           keySpecG = new SecretKeySpec(sKey.getEncoded(), "AES");
       }

       // Initialization Vector generating method
       public static void ivGen() {
           SecureRandom secureRandom = new SecureRandom();
           // Creates a new instance of secure random that uses the
               systems source of entropy to generate random numbers

           byte[] iv = new byte[16]; // Creates a new 16 byte iv ->
               standard iv size for aes
           secureRandom.nextBytes(iv); // fills the 16 bytes of the iv
                with a secureRandom number
           // Creates a new ivSpecification to be used with the
           // cipher class in the encryption and decryption methods
           ivSpecG = new IvParameterSpec(iv); // Saves the iv Spec to
               the global variable so that it can be used for both
               operations
       }

       public static String encrypt(String data)
```

```
            throws NoSuchAlgorithmException, NoSuchPaddingException
                ,
            InvalidAlgorithmParameterException, InvalidKeyException
                ,
            IllegalBlockSizeException, BadPaddingException {

        IvParameterSpec ivSpec = ivSpecG; // Generates the IV using
             the method above
        SecretKeySpec sKeySpec = keySpecG; // Generates the key
            using the method above

        // Creates a new instance of the cipher class to be used to
             encrypt the data using the AES cbc standard
        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING");
        cipher.init(Cipher.ENCRYPT_MODE, sKeySpec, ivSpec); //
            Initializes the cipher with the required mode, key and
            iv

        byte[] encData = cipher.doFinal(data.getBytes(
            StandardCharsets.UTF_8)); // Encrypts the data with the
             cipher
        return Base64.getEncoder().encodeToString(encData); //
            Returns the encrypted string in base64
    }

    public static String decrypt(String data)
            throws NoSuchPaddingException, NoSuchAlgorithmException
                , InvalidAlgorithmParameterException,
            InvalidKeyException, IllegalBlockSizeException,
                BadPaddingException {

        // Does the exact same thing in reverse to the encrypt
            method
        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5PADDING");
```

```
            cipher.init(Cipher.DECRYPT_MODE, keySpecG, ivSpecG); //
                This time we set the mode to DECRYPT
            byte[] decoded = cipher.doFinal(Base64.getDecoder().decode(
                data)); // Decrypts the data from base 64 and then
            // from the aes encrypted ciphertext back into a string
            return new String(decoded); // returns a string of the
                decoded aes ciphertext
        }
    }

    // AES Testing method
    public static void testAES() {
        System.out.println("Testing␣AES␣.␣.␣.");
        // Creates a new instance of the AES class
        AES aes = new AES();
        for (int i = 0; i < 10; i++) {
            System.out.println("Pass␣" + (i + 1) + ":␣Running␣.␣.␣.");
            try {
                // Generates the iv used for the AES encryption
                aes.ivGen();

                // Times the key generation for the AES algorithm
                long s = System.currentTimeMillis();
                aes.keyGen();
                long st = System.currentTimeMillis();
                long t = st - s;
                keyTimeAES.add(t);

                // Times the encryption of the random plaintext String
                s = System.currentTimeMillis();
                String ciphertext = aes.encrypt(randomStrings.get(i));
                st = System.currentTimeMillis();
                t = st - s;
                encTimeAES.add(t);
                // Times the decryption of the ciphertext
                s = System.currentTimeMillis();
                String message = aes.decrypt(ciphertext);
                st = System.currentTimeMillis();
                t = st - s;
                decTimeAES.add(t);

                // Performs string verification check to ensure that
                    the input data matches the decrypted string
                if (randomStrings.get(i).compareTo(message) == 0) {
                    validAES.add(Boolean.TRUE);
                    System.out.println("Pass␣" + (i + 1) + ":␣
                        Successful!");
                } else {
                    validAES.add(Boolean.FALSE);
                    System.out.println("Pass␣" + (i + 1) + ":␣Failed␣->
                        ␣Input␣String␣mismatch␣with␣decrypted␣data!");
                }

            } catch (NoSuchAlgorithmException |
                InvalidAlgorithmParameterException |
                NoSuchPaddingException | IllegalBlockSizeException |
                BadPaddingException | InvalidKeyException e) {
                e.printStackTrace();
            }


        }
    }

    // RSA Testing method
    public static void testRSA() {
        System.out.println("Testing␣RSA␣.␣.␣.");
```

```
// Creates a new instance of the RSA class
RSA rsa = new RSA();

for (int i = 0; i < 10; i++) {
    System.out.println("Pass␣" + (i + 1) + ":␣Running␣.␣.␣.");
    try {

        // Times the key generation for the AES algorithm
        long s = System.currentTimeMillis();
        rsa.keyGen();
        long st = System.currentTimeMillis();
        long t = st - s;
        keyTimeRSA.add(t);

        // Times the encryption of the random plaintext String
        s = System.currentTimeMillis();
        String ciphertext = rsa.encrypt(randomStrings.get(i));
        st = System.currentTimeMillis();
        t = st - s;
        encTimeRSA.add(t);

        // Times the decryption of the ciphertext
        s = System.currentTimeMillis();
        String message = rsa.decrypt(ciphertext);
        st = System.currentTimeMillis();
        t = st - s;
        decTimeRSA.add(t);

        // Performs string verification check to ensure that
            the input data matches the decrypted string
        if (randomStrings.get(i).compareTo(message) == 0) {
            validRSA.add(Boolean.TRUE);
```

```
                System.out.println("Pass␣" + (i + 1) + ":␣
                    Successful!");
            } else {
                validAES.add(Boolean.FALSE);
                System.out.println("Pass␣" + (i + 1) + ":␣Failed␣->
                    ␣Input␣String␣mismatch␣with␣decrypted␣data!");
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}


// Plaintext String Generator of given integer length
public static String randomPlaintextGen(int length) {

    String upper = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    String lower = upper.toLowerCase(Locale.ROOT);
    String decimalDigits = "1234567890";
    String acceptedChars = upper + lower + decimalDigits;

    Random random = new Random();
    StringBuilder out = new StringBuilder(length);

    for (int i = 0; i < length; i++) {
        out.append(acceptedChars.charAt(random.nextInt(
            acceptedChars.length())));
    }

    return out.toString();

}
```

```
// Generates and stores randomly generated plaintext strings
public static void fillRandomPlaintext() {
    // Random strings generation
    int length = 0;
    for (int i = 0; i < 10; i++) {
        length += 50;
        randomStrings.add(randomPlaintextGen(length));
    }
}

// Saves an output file with the statistics from testing the
   algorithms
public static void summary() {

    try {
        writeLinkedListToFile("keyTimeAES", keyTimeAES);
        writeLinkedListToFile("encTimeAES", encTimeAES);
        writeLinkedListToFile("decTimeAES", decTimeAES);

        writeLinkedListToFile("keyTimeRSA", keyTimeRSA);
        writeLinkedListToFile("encTimeRSA", encTimeRSA);
        writeLinkedListToFile("decTimeRSA", decTimeRSA);
    } catch (Exception e) {
        e.printStackTrace();
    }



}

public static void writeLinkedListToFile(String fileName,
   LinkedList list) {
    try {
        // https://stackoverflow.com/questions/24982744/printwriter
           -to-append-data-if-file-exist
        File file = new File("outputs/" + fileName + ".txt");
        PrintWriter printWriter = null;
        if (file.exists()) {
            printWriter = new PrintWriter(new FileOutputStream(file
                , true));
        } else {
            printWriter = new PrintWriter(file);
        }
        printWriter.println(list.toString());
        printWriter.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public static void runTests(int tests) {
    for (int i = 0; i < tests; i++) {
        // clears all the lists ready to run another test
        if (i > 0) {
            randomStrings.clear();

            keyTimeAES.clear();
            encTimeAES.clear();
            decTimeAES.clear();
            validAES.clear();

            keyTimeRSA.clear();
            encTimeRSA.clear();
            decTimeRSA.clear();
            validRSA.clear();
        }
        // Fills the plaintext array with the random strings
        fillRandomPlaintext();

        // Tests the AES algorithm
        testAES();

        // Tests the RSA algorithm
        testRSA();

        summary();
    }

    // print out description of what is in each output file
    System.out.println("Performance␣Specifications␣−>␣milliseconds"
        );
    System.out.println("Key␣Generation␣Times:␣");
    System.out.println("Encryption␣Times:␣");
    System.out.println("Decryption␣Times:␣");
    System.out.println("Valid␣Encrypts␣&␣Decrypts:␣");

}

public static void main(String[] args) {

    runTests(20);

}
}
```

6.2 Raw Data

AES Key Generation Times - Milliseconds										
	Data Set Size in Characters									
Run Number	**50**	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
1	39	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	1	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	1	0
18	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0

Figure 6.1: AES Key Generation Times Raw Data

AES Encryption Times - Milliseconds										
	Data Set Size in Characters									
Run Number	**50**	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
1	3	0	0	0	0	1	0	0	0	1
2	0	0	0	0	1	1	0	1	0	0
3	0	0	0	0	1	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	1	0	0	1	0	0
6	0	0	0	1	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	1	0	0	0	0	1	0	0	1
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	1	0	0	0	0	1
12	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0
14	1	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	1	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0
19	1	0	0	0	0	0	1	0	0	0
20	0	0	0	0	0	0	0	0	0	0

Figure 6.2: AES Encryption Times Raw Data

AES Decryption Times - Milliseconds										
	Data Set Size in Characters									
Run Number	**50**	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
1	0	0	1	2	0	0	1	0	0	0
2	0	0	1	0	0	1	1	0	1	1
3	0	0	0	0	0	0	0	0	0	0
4	0	1	0	0	1	0	0	1	0	1
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	1	0	0	0	0	0
9	0	0	0	1	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0
15	1	0	0	0	0	0	0	1	0	0
16	1	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	1	0	0
19	0	0	0	0	0	0	0	0	0	0
20	0	0	1	0	0	0	0	0	0	0

Figure 6.3: AES Decryption Times Raw Data

RSA Key Generation Times - Milliseconds										
Run Number	**Data Set Size in Characters**									
	50	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
1	1332	1707	3728	1608	564	2383	1120	471	841	1186
2	1003	1864	943	1120	740	682	2639	554	1491	771
3	1088	889	797	1042	1443	554	1382	1285	1494	1167
4	279	4820	2025	1288	704	926	789	869	325	2551
5	1313	3761	637	2004	747	2772	1685	2457	690	1121
6	1366	787	1373	1163	523	1026	1287	1460	1618	1119
7	1483	1563	2900	2326	925	1082	1027	408	552	1386
8	2486	249	395	468	1687	400	2529	320	1003	3553
9	754	1093	3628	2181	1056	469	1451	210	1921	1031
10	1157	992	1360	1347	1127	1691	2118	597	730	1477
11	1285	1613	894	438	1871	518	840	1464	2028	3637
12	2116	321	2737	2010	927	3299	1074	418	2243	725
13	566	1666	816	2150	1651	2746	2782	1662	680	2306
14	2055	855	1694	243	319	460	1309	2438	244	738
15	3027	672	1305	3459	1036	569	1077	1495	2650	585
16	199	1031	709	666	2444	1279	1292	1828	1129	460
17	912	3204	2058	2567	613	2343	725	410	917	2109
18	2475	1438	1215	871	1317	2020	1693	673	1873	2106
19	1333	1684	447	848	774	3133	833	559	833	4479
20	1611	436	536	645	925	2435	1021	1916	1210	787

Figure 6.4: RSA Key Generation Times Raw Data

RSA Encryption Times - Milliseconds										
Run Number	**Data Set Size in Characters**									
	50	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
1	1	1	0	1	1	1	0	1	0	1
2	0	1	0	0	0	1	1	1	0	0
3	0	0	0	1	0	1	0	0	0	0
4	1	1	0	0	0	1	0	1	1	1
5	1	1	0	0	0	0	0	0	0	0
6	0	0	0	1	0	0	0	1	1	0
7	0	0	0	0	1	0	0	1	0	0
8	0	0	0	0	1	0	0	1	1	0
9	0	0	0	0	1	0	0	1	0	0
10	0	1	0	1	0	1	0	0	0	0
11	0	0	0	0	0	1	1	0	0	0
12	1	0	0	0	0	0	0	0	0	0
13	0	0	1	0	0	0	0	0	0	0
14	0	1	1	0	1	1	0	0	0	0
15	0	0	1	1	0	0	0	0	0	0
16	0	0	1	0	0	0	1	1	1	0
17	1	0	0	1	1	0	0	1	1	0
18	0	1	0	0	0	0	0	0	0	0
19	0	1	0	0	0	1	1	1	0	1
20	0	0	0	1	0	1	0	0	1	1

Figure 6.5: RSA Encryption Times Raw Data

RSA Decryption Times - Milliseconds										
Run Number	**Data Set Size in Characters**									
	50	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
1	13	27	29	10	10	11	10	10	12	13
2	12	14	10	10	10	9	10	10	11	11
3	9	10	11	9	10	11	10	10	10	10
4	11	11	10	10	10	9	10	11	10	9
5	9	12	12	10	10	10	10	10	10	10
6	10	11	11	10	11	10	10	10	14	10
7	10	10	10	10	10	10	11	9	10	11
8	10	10	10	10	9	10	10	9	11	10
9	10	10	10	10	10	11	10	11	11	10
10	10	10	10	10	10	10	11	10	11	10
11	10	10	10	10	10	9	9	10	11	11
12	9	11	10	0	16	17	0	15	21	0
13	0	10	9	10	10	10	10	11	10	10
14	9	9	9	11	9	9	10	9	10	10
15	10	11	11	18	10	10	11	10	10	10
16	11	10	9	11	10	10	9	9	9	10
17	16	10	10	9	10	11	10	9	14	10
18	11	10	10	10	9	10	10	11	10	10
19	10	9	10	10	10	10	9	9	10	9
20	11	10	10	9	10	9	10	10	9	9

Figure 6.6: RSA Decryption Times Raw Data