An Investigation into the use of Rijndael Encryption in Various Computing Situations

Thierry Draper
BSc (Hons) in Computer Science

2005

An Investigation into the use of Rijndael Encryption in Various Computing Situations

Submitted by Thierry Draper

Copyright

Declaration

Signed: ....

Abstract

The algorithm specified in the Advanced Encryption Standard, Rijndael, has been implemented in six different programming languages and compare to each other, with respect to run-times and conclusions drawn. These tests were performed using the three key sizes specified in the Standard and applied to both encryption and decryption. It was found that the C and Java implementations the quickest, followed by the Perl and JavaScript versions and finally, the Common Lisp and Matlab implementations were the slowest.

Acknowledgements

Firstly, I would like to thank Dr Russell Bradford for his supervision of this project.

I would also like to thank Family and Friends for their support throughout the duration of this project, specifically David Mayo, whose advice and experience was extremely helpful. In addition, Helen Chilcott, who kept me fed and watered during the course of this dissertation, whose patience whilst teaching me Statistics was unending and her use of red pen was worse than my English teacher!

Contents

Chapter 1

Introduction

In 1976, the National Bureau of Standards (now the National Institute of Standards and Technology), adopted as a federal standard, the Data Encryption Standard, which is also known by its abbreviation — the DES. DES was a standard that introduced a new cryptographic algorithm (the Data Encryption Algorithm, or DEA). DES revolutionised the cryptographic community there had not been a publicly released cryptographic algorithm since World War I — and reinvigorated public and academic interest in cryptography. However, in the late 1990s, DES was nearly 20 years old and was replaced by a new standard, the Advanced Encryption Standard (AES). Specified as part of this new standard was a new encryption algorithm, the Advanced Encryption Algorithm, which was called Rijndael by its creators.

The purpose of this investigation is to assess the suitability of Rijndael with respect to various software implementations in different programming languages. This suitability will be tested by obtaining performance figures of the algorithm in each language and comparing relevant results using appropriate statistical methods.

Chapter 2 looks at the history of Rijndael and why it was chosen over other candidates as the Advanced Encryption Standard. The languages being used will also be considered and how they have been previously applied to the field of cryptography.

After reviewing the history of Rijndael, Chapter 3 explores in detail, at how the finer points of the experiments were to be carried out, by discussing the various aspects of the problem.

Once the experiments had been designed, it was necessary to implement the algorithm and their applications; Chapter 4 examines the particulars of this implementation process.

Chapter 5 explores how the various aspects of the implementations were tested to ensure they worked correctly, as well as how some of the errors produced were overcome.

After creating the implementations and testing them to ensure their correctness, it was necessary to run the experiments. The whole process is described in Chapter 6, along with the important data that was acquired.

Once this data had been accumulated, Chapter 7 discusses the statistical methods that were utilised to allow some conclusions to be drawn from the raw data, whilst illustrating this method on one of the experiments.

At the completion of the analysis, Chapter 8 looks back at the whole investigation, detailing the achievements of the project, as well as critically evaluating how the project was carried out.

Finally, the Appendices contain the parts of the investigation that are necessary for this project, but not relevant to be included in the main body of this document. These include, the remaining Statistical Analysis that was performed on the data from Chapter 6 in Appendix A and the Code Listings for the implementations in Appendix B.

Chapter 2

Literature Survey

2.1 Introduction

This project proposes to develop multiple software applications, utilising the Rijndael encryption algorithm in a variety of programming languages. There are two purposes of this literature survey: to help get a better idea of the problem being investigated and to obtain more information about the algorithm. It is useful to capture a better idea of the problem and so clarify the important aspects of the project, but it is also imperative that a further understanding of the algorithm, its comparative merits (in relation to the other AES candidate algorithms) and the way it works are obtained before the project is commenced. To do this, it would be useful to read into the history and selection process of the Advanced Encryption Standard (AES), since this is where Rijndael is most commonly associated.

It is also necessary to research the inner workings of the algorithm, so that the usefulness of languages learnt during the first two years of University could be assessed for suitability. If not, it would be necessary to research which languages would need to be learnt so that an adequate number of applications are developed to make the project worthwhile.

2.2 Advanced Encryption Standard and Rijndael

In 1997, NIST announced, in both [16] and [17], a call for ideas from the cryptographic community regarding a new encryption standard to replace the current standard - the Data Encryption Standard. The DES had initially been approved in 1976, (later reaffirmed in 1983, 1988, 1993 and 1999) [13], but NIST had decided during the 1993 review that as part of the 1998 re-evaluation, a successor to the DES may be proposed as the DES would be over 20 years old [14] [28].

Although DES was historically important in the previously secretive field of cryptography [25], it was starting to age. As early as the latter parts of the 1980s, it was widely assumed that DES was going to be broken in the near future, even though it was thought that cryptanalysts were still a long way off cracking it. During the 1988 review, NIST decided that a new algorithm to take the place of DES should be defined and implemented as a new standard. Although this took nearly a decade, in 1997 NIST announced it was looking for DES's replacement — the Advanced Encryption Standard (AES) [17]. The AES was to specify a publicly available encryption algorithm, usable worldwide for free, capable of protecting sensitive government information well into the future.

This initial announcement was made for members of the public to contribute ideas and comments regarding the draft submission requirements of the AES, which was to be published by NIST. These comments were then discussed in a public workshop, as well as in private. Later that year, NIST announced its request for candidate algorithms to be used as part of the Advanced Encryption Standard [13].

This notice outlined the basic features of each submitted algorithm, but left a lot of freedom for submitters to design their algorithms. NIST was looking for a block cipher to be used, although if another method was used and had been shown to be advantageous enough, it would still be considered. It had been decided that the algorithm should employ a symmetric secret key cipher and *at the very least* support 128-, 192- and 256-bit key sizes, with 128-bit blocks. [27].

21 algorithms were submitted, but only 15 satisfied the "complete and proper" prerequisite set out in [13].

2.2.1 Candidate Algorithms

NIST received 15 algorithms that satisfied the requirements and thus were declared to be usable as the standard. These algorithms had been classified by design paradigm [3] (cited in [19]) to show how each candidate algorithm had been influenced.

Naturally, the main selection criteria during the first round was security. However, secondary consideration regarding cost (both computation resources and licensing) and algorithm characteristics (such as flexibility with key/block sizes and ease of implementation across platforms) was also applied.

2.2.2 Why Was Rijndael Chosen?

After the first round analysis of the 15 candidate algorithms [19], five algorithms were left: MARS, RC6, Rijndael, Serpent and Twofish. These algorithms were spread evenly among the design paradigms stated in [3] (cited in [19]), showing that an algorithms influence was not important.

These five algorithms were then scrutinised further, again both publicly and by NIST. As the next stage of the selection process was the selection of the final algorithm, each algorithm was inspected in much more detail, in a variety of aspects, ranging from security to versatility and flexibility [18].

Security Concerns

Each algorithm passed the first round, in part because of a lack of major security concerns. The only attacks noted during the public analysis period was purely when each algorithm was simplified by reducing the number of rounds used in the encryption process. This was considered a sign of potential weakness, but was not considered major due to the actual number of rounds used in the submitted algorithms.

NIST tried to quantify some of the security properties of each candidate, but this was quite an awkward task and does not produce concrete results. However, one such feature NIST considered was the idea of *security margin* (the number of rounds in the algorithm vs. the minimal number needed for security), originally introduced in [3]. Although each candidate had an adequate security margin, that of MARS, Serpent and Twofish showed higher margins than RC6 and Rijndael.

Software Implementations

Software implementations of each algorithm are dependent on many external factors, such as the hardware on which the implementations are run and the compiler used to compile the provided source. However, for the purposes of the analysis, this was not considered too large a problem since these factors would be applied to each candidate and there were numerous hardware/software combinations used to analyse the algorithms. For instance, two of the algorithms, MARS and RC6, use slightly more complicated operations than the other three finalists, so some the test results depend more greatly on the above factors. The algorithms performed well when using hardware/compiler combinations which support these complex operations, whilst they did not perform as well when run on hardware/compiler combinations without these complicated instructions.

Generally, Twofish did not execute well in the performance tests, mainly because the algorithm is designed towards the secure end of the security/time trade-off. As expected, MARS and RC6 had average performances across all platforms, but performed better where the more complex operations were intrinsic. Serpent was generally the slowest, although it was consistent for both encryption and decryption. Rijndael performed consistently well across all platforms, although the encryption/decryption time increases when the key size increased.

Hardware Implementations

Due to the increased degrees of freedom in designing a hardware implementation compare to that of a software implementation, it was more difficult to analyse each algorithm in hardware terms. To analyse the hardware, NIST used two major classes — Application Specific Integrated Circuits and Field Programmable Gate Arrays. Whilst this did not provide the abundance of data that was generated by the software implementation tests, enough data was provided to perform the analysis of hardware use on each of the candidate algorithms.

Rijndael and Serpent had the best hardware throughput, although Rijndael's throughput decreased when the larger key sizes were used. RC6 and Twofish had an average throughput, but both can be implemented compactly. Due to an above average area requirement (and hence cost) and consistently below average efficiency, MARS was not considered particularly suitable for hardware use.

Attacks on Implementations

Section 2.2.1 discussed the security of the algorithm in terms of mathematical "soundness". However, it is also possible to attack each algorithm by the physical characteristics of its implementation. One such attack is based on the amount of time and/or power the implementation uses. A Timing Attack is based on the execution time of various operations, depending on their arguments, whilst a Power Analysis Attack is based on the power consumption pattern of each operation, which again, could depend on the operation's arguments. This implies some of the candidate algorithms may be more at risk in certain environments than others, however, it is possible to defend against these attacks.

The main defence against a Timing Attack is to fix the run-times of the encryption and the decryption processes to run in as similar an amount of time as possible. Software balancing can be used as a defence against Power Analysis, by masking the power consumption of certain operations which is achieved by executing these operations multiple times and then re-merging the results back together.

Undefended, Rijndael and Serpent use operations which are among the easiest to defend against, whilst MARS and RC6 use complex operations, such as 32-bit multiplication, which are difficult to defend against. Twofish, however, only has the one complex operation, which is moderately difficult to defend against.

Whilst defending against these sorts of attacks, Rijndael, Serpent and Twofish have less of a performance hit than MARS and RC6. In fact Rijndael even has a slight speed advantage when such protections are used!

Restricted-Space Environments

Although not an explicit requirement of the original algorithm request [13], during the second round of evaluation, NIST wanted to see how the algorithms performed when implemented on smartcards, where memory is at a premium. The amounts of ROM and RAM necessary for an implementation to run in a restricted-space environment may also be a factor in deciding the suitability of the algorithm in certain situations.

Rijndael, Serpent and Twofish all appear suitable due to their low ROM and RAM usage, although Rijndael and Serpent require extra ROM if both encryption and decryption are to be implemented on the same device. RC6 has low ROM requirements but, due to needing to store subkeys in memory, has a high RAM requirement. MARS is not suitable for use in restricted-space environments due, not only to its high ROM requirements, but its use of pattern-matching, which in such environments, requires extra resources.

Encryption versus Decryption Functions

Another consideration with restricted-space environments is the physically available space. In certain situations, only encryption or decryption may be necessary and so these space considerations are not a worry. However, if both encryption and decryption are necessary and there is little available physical space, drastically different encryption and decryption functions may be of concern.

MARS, RC6 and Twofish all use similar, if not identical encryption and decryption functions and so require little additional hardware area to implement decryption along with encryption. Rijndael's encryption and decryption functions are significantly different and so require more area, but can be implemented to share hardware resources. Serpent also has considerably different functions, but is unable to share such resources and so requires a larger area again.

Key Agility

Key schedule computation can take one of two forms: pre-computation or on-the-fly generation (computation of required subkeys just before use in a round). Key agility is affected by subkey computation, since if pre-computed, more memory is required, which could be a problem if utilised in a restricted-space environment.

Rijndael, Serpent and Twofish support on-the-fly subkey generation for both encryption and decryption, although Rijndael requires the entire key schedule to be computed before the first round of decryption. MARS has similar characteristics to that of Rijndael's, but 10 of the 40 subkeys must be computed and stored at the same time, placing an additional burden on the implementation. RC6 only supports on-the-fly subkey computation for encryption, not decryption, so subkeys necessary for decryption must be pre-computed.

Key and Block size flexibility

Although the original requirement of the successful algorithm request [13] was for the algorithm to be capable of handling 128-, 192- and 256-bit key sizes with a 128-bit block size, the call [13] also pointed out that an algorithms ability to handle additional key and block size combinations would be considered during the evaluation. The future benefits of such a flexible algorithm are obvious. For instance, if the NIST decide to change the standard (e.g. in the case of newly identified attacks), then a whole new algorithm is not necessary as the key and block sizes can be increased.

RC6 and Rijndael include the greater support for arbitrary key and block sizes, much beyond the 256-bit required. MARS supports arbitrary key sizes between 128- and 448-bit, but is not as flexible as either RC6 or Rijndael. Serpent and Twofish support arbitrary key sizes up to 256-bits, but like MARS, is not as flexible.

Conclusion

Whilst inspecting the above test results, it became apparent that each algorithm offered adequate security alongside numerous advantages. However, each algorithm had one or more field which was weaker than some of the other algorithms and as such, there was no clear-cut candidate which would have been an obvious choice.

After inspecting the sections described above, Rijndael was selected as the Advanced Encryption Algorithm for the Advanced Encryption Standard based on a “combination of security, performance, efficiency, ease of implementation and flexibility” [15]. More specifically, Rijndael was a consistently good performer in both hardware and software tests across a range of environments; required small amounts of memory making it suitable for restricted-space environments; utilises operations which are the easiest to defend against power and timing attacks; has excellent key setup time and agility; highly flexible key and block sizes, as well as the ability to alter the number of rounds used in the algorithm.

2.3 About Rijndael

Rijndael, was created by two Belgian cryptologists, Dr Joan Daemen and Dr Vincent Rijmen. Rijndael was based on a previous cryptographic algorithm called Square, which was created by the same designers. Although Rijndael and Square ([5]) were designed by the same people and share some common aspects, they are two different algorithms [7].

Rijndael was designed with simplicity and modularity in mind [22]. The simplicity is brought about by avoiding any complex operations (which, as indicated in section 2.2.2, improves security) and using a simple, yet quick key schedule. This not only improves performance of the encryption process but also that of the decryption process.

The first aspect that should be noted about the Rijndael algorithm is that some of the operations are performed over a Galois Field, GF(2^8) [6]. This finite mathematical field ensures that the product of two numbers does not exceed a certain number of digits. This is important in Rijndael because the algorithm uses multiplication as one of its operations and the resulting answer cannot exceed the 8-bits in a byte. Although there are other intricacies in Galois Fields which are used in the algorithm, these are dealt with in the construction of the S-Boxes. As these S-Boxes are fixed for all implementations (regardless of input or key), it will be possible for these tables to be coded using fixed values.

The algorithm's authors also briefly indicated the security of the algorithm, by comparing the number of rounds used with the highest known attack of a similar type of algorithm (the idea of *security margin*, originally described in section 2.2.2). Although this was not as high as other candidate algorithms, this demonstrated the security of the algorithm [23].

2.4 Rijndael Specification

Rijndael is a block cipher; that is, a cipher which splits the original input into blocks and encrypts these blocks separately, to create the encrypted output. Rijndael works with blocks of 128-bits and outputs blocks of the same length. The algorithm also uses a key (which can be of 128-, 192- or 256-bit length) to ensure it is possible to encode the same plain text differently each time the algorithm is run (of course, assuming a different key is used).

The algorithm starts by converting the input into an internal 'State', which is manipulated by the algorithm. After this, the original Cipher Key is recursively lengthened into an 'Expanded Key', the length of which is dependent on the Block size and how long the algorithm needs to run.

After Exlusive-ORing the first 128-bits of the Expanded Key with the State, a series of transformations (called 'Rounds') take place on the State, the number of which depends on the size of both the key and input block. A Round consists of the following transformations:

- Byte Substitution - Each element of the State is copied for another (based on value) in a pre-computed substitution table (more commonly called the 'S-Box').

- Row Shifting - Each row of the State is cyclically shifted, the size of which is dependent on the row and block size.
- Column Mixing - Each column is multiplied by a fixed 4×4 matrix.
- Key Application - Part of the Expanded Key is Exlusive-ORed with the State.

Finally, one more round similar to that of the above, is applied, although this time without the Column Mixing. To finish the cipher process, the State is collapsed back into a single dimension array and returned, to be used by the application as necessary.

To keep the multiplication a constant length (i.e. each cell of the State at 8-bits), it is necessary to perform this multiplication in a Galois Field [12] (cited in [6]), rather than the classical field in Mathematics. In GF(2^8), all multiplication is performed modulo a 9-bit prime number (100011011, or '11b' in hexadecimal) to ensure that the resulting value does not exceed 8-bits (so can still fit in a byte). Addition is performed by the Exclusive-OR operator.

The above describes only the cipher — the inverse cipher also needs to be explained, where the original plain text is obtained from the cipher text, given the same key. The inverse cipher was designed to be identical in structure to the cipher, although naturally, certain aspects need to be changed.

The aspects of the inverse cipher that are different from the cipher are:

- After the Key Expansion, an application of the (inverse) Column Mixing transform is applied;
- The subkeys are applied backwards (starting at the end of the expanded key to the start as the rounds progress);
- The Byte Substitution table is replaced with a different S-Box of the same size (but works in the same way);
- The matrix in the Column Mixing step is replaced with a different 4×4 matrix;

2.5 Software Implementations of Rijndael

The above provided an overview of the algorithm's origins and details, but does not cover how the algorithm will be used in this investigation. Now that a basic grounding of the algorithm has been established, it is time to decide which languages will be used in this investigation and the applications which will demonstrate each implementation.

Ideally, the project should create between four to six different applications to illustrate the Rijndael algorithm. The obvious application would be with the cryptographic handlings of a file, for which a pair of linked applications will be developed: one for encrypting files, the other for decrypting the previously encrypted files. It would be sensible to use ANSI C for the encryption of binary files since the language's capabilities of such a problem are well known. To handle text files, ANSI C would again be suitable, but it would be more preferable to instead use a different language for this particular case. Perl, a general purpose language (much like ANSI C), which can be used to generate scripts to run on Unix or Linux, would be an appropriate language for this purpose.

Another possible use for the algorithm would be the encoding of passwords used on a website. For security reasons, the cipher text password would be transmitted between the client and the server, rather than the plain text. It would be necessary to encrypt the password before transmission, therefore JavaScript can be used for this. Another web-based application would be sending a secure email via a webpage, for which a Java applet would be an ideal choice of implementation.

To attempt to create a more interesting project, Matlab and Common Lisp implementations of the Rijndael algorithm will be developed. Due to the nature of these languages, applications need not be developed, as the functions will be available for use in the run-time environment. These are not necessarily obvious choices of languages for such a project, but for this same reason, the Matlab and Common Lisp implementations would produce interesting results. Though these may be complicated implementations, the Rijndael Homepage [21] contains links to other people's work, including links to Matlab and EMACS Lisp implementations. Although EMACS Lisp is different to Common Lisp, this implementation shows it is possible to encode Rijndael in a dialect of Lisp; although the dialect differences need to be taken into consideration. (A search using Google listed [35] and [8], further showing it is realistic to create such an implementation, although one is not linked in [21]).

2.6 Language Uses

The final consideration before the design stage of the project can be started is how each of the chosen languages has been used in cryptography. The more general purpose languages, like C and Java, have a rich history in cryptography — not only generally, but also with AES as submissions had to include C and Java implementations. There have been many books published on cryptography, many of which (including [25]) use C code to demonstrate the various algorithms.

Java is also a popular choice for cryptography. Since Java is a widely favoured language for creating Web based applications (through applets), it has an obvious need for cryptography. As such, much effort has gone into creating necessary libraries to help with this. An example of this is the extension to the standard Java SDK, called the Java Cryptography Extension [31], which allows Java programmers to use cryptography in their applications with relative ease. Many books have discussed this, and more generally cryptography in Java, such as [4] and [37].

Perl is a widespread, stable programming language, which can be equipped with many modules to aid developers when creating their Perl scripts. One category of these modules is in the field of cryptography, for which there are many available modules. These modules offer a variety of cryptographic algorithms, but some (including [26]) use Rijndael itself.

JavaScript is widely used to protect data in Web Browsers between form submission and the data being transmitted through HTTP, where it may be intercepted and be viewed by anyone, if not encrypted. Naturally, it would be beneficial if the data was encrypted before transmission. When dealing with web browsers, JavaScript is a highly suitable language, for amongst other reasons, it is used widely in Web Browsers. Online JavaScript examples (including a slightly deeper insight into why JavaScript is used in cryptography) can be found in [36].

Among others, Matlab is designed for data visualisation, data analysis and numerical computation. Additionally, it provides the necessary bitwise functions to perform Rijndael, but has been mainly used in cryptography with Public Key Encryption (such as RSA, [24]). RSA does not use bitwise functions like Rijndael, but instead its security lies in its numerical manipulation — something to which Matlab is well suited. Further to this, there has been a book published ([33]), which uses Matlab to illustrate the Mathematical side of Cryptography, including discussions on DES and RSA.

Finally, Common Lisp is a popular language for coding by many developers because of its functional nature. Although its use is not as widespread as some of the other languages in consideration, it is still used in certain areas of programming and many different cryptographic algorithms have been illustrated in [34].

2.7 Summary

This literature review has been a concise description of the Rijndael algorithm and its place in the search for an algorithm as the Advanced Encryption Standard (AES), as set by the National Institute of Standards and Technology (NIST). Firstly, AES was discussed and its place with regards to similar standards identified. Then, the narrowing of the search was examined, with the reduction of the 15 candidate algorithms down to five, before the pros and cons of each of the five finalists were established for each of the selection criteria. The reasons for the selection of Rijndael as the Advanced Encryption Algorithm for the AES were then established. The second part of this review was more concerned with the Rijndael algorithm; its main characteristics and details were discussed to give a better understanding of the types of languages which would be suitable for this project.

Chapter 3

Design of Experiments

3.1 Design Choices

This is an important part of the project, where it will be specified how comparisons will be drawn between the various implementations. As there are many varying factors in the execution of each implementation, it would be preferable to fix as many of these as possible. Fixing these factors reduces the ambiguity as to where the differences in each implementation occur. For instance, if one implementation runs significantly quicker than another, it is preferable to reduce the number of reasons as to why this may occur.

3.1.1 Complexity of Applications

It is first necessary to discuss the applications that will illustrate the use of the Rijndael algorithm. These applications are to be basic applications which should be only so complicated as is strictly necessary — the project is concerned with the implementation of the algorithm and not the application, so the application should not include anything that is not required within the algorithm.

This comment is mainly aimed at the Java and JavaScript applications, where it is clear that unnecessary work could easily be carried out. For instance, the Java email input screen could contain many fields which are not required in the encryption process (such as options for Read/Delivery Receipts, Option to save messages to a draft folder), although a 'To:' field would be necessary to at least ensure the interface looks appropriate (although the 'To:' field is ignored when encrypting).

3.1.2 Implementation Type

An important design factor, which could have a drastic impact on the run-time of the algorithm, is the way the implementation is coded. Since it is possible to implement Rijndael in two very different styles, it was necessary to decide which of the two methods would be used. It is, of course, more preferable to use a single method for coding all the implementations in this investigation since it reduces one of the variable factors mentioned above. If one implementation runs, on average, quicker than a second implementation (which is implemented using the alternative style), is this due to differences in the language or because the two implementations were implemented differently? It is for this reason, that one of the two methods must be chosen.

One of the styles is to implement the algorithm in the same way as it was explained in [6] and described in section 2.4. This method converts the input into 4 blocks of 4 bytes (the ‘State’) and then each round manipulates each byte of each column of this State individually. Each round is performed by using various sub-routines used to manipulate each byte in a column using substitutions (using the S-Box), various operators (such as XOR), Galois Field multiplication (in GF(2^8)) and application of a key (which is derived from and expands the original Cipher Key).

The other style is to use table lookups, where each round is condensed into a series of table lookups on pre-computed tables. Similar to the set up for the traditional algorithmic method above, the input is converted into a number of blocks of 4 bytes (32-bit ‘words’) and each round of the algorithm consists of a series of table lookups on each column in the State, followed by an application of the expanded Cipher Key.

It was decided that the best approach for this project would be to use the first style suggested — the traditional ‘algorithmic’ design, rather than the using the Table Lookup method. The was chosen because, although the table lookup method is quicker, the ‘algorithmic’ design is easier to read and understand. The absolute run-time values of each implementation are not of concern, only the relation between them, so the extra speed gained by the tabular design is outweighed by the readability of the ‘algorithmic’ design.

3.1.3 Cryptographic Mode

There are two basic block cipher modes, which can be used to handle the blocks during the encryption/decryption process — Electronic Cookbook mode (ECB), or Cipher Block Chaining mode (CBC). These modes do not have any direct impact on the algorithm, but instead with how the blocks are used before encryption and after decryption, so lie in the application.

Electronic Cookbook mode is the simplest way of implementing a block cipher — the input is split into blocks of appropriate size. The resulting encrypted/decrypted output is simply the result of encrypting/decrypting the blocks and putting them back together in the correct order. Although this is the easiest to implement, it does have two inherent security risks.

The most obvious of these risks is illustrated when a single key is used multiple times; given any key, any plain text will *always* encrypt to the same cipher text. This is most evident at the start and end of some messages, when standard header/footers are used (this is called 'Stereotyped Beginnings' and 'Stereotyped Endings'). If a cryptanalyst is able to obtain this information, then their job becomes slightly easier.

However, possibly less obvious but a considerably more serious risk, is that of Block Replay. Block Replay is when an intruder intercepts a message of a known format and switches some of the blocks with the blocks obtained from a previous message for the purpose of fooling the receiver.

The example given in [25] is of a bank deposit — Alice (the deposit sender) wishes to pay Bob (the receiver) via a money transfer system. However, Mallory (a greedy third party) can switch the appropriate blocks of the transfer with his own details (obtained from a previous money transfer) to get Alice's money instead! Naturally, Bob will eventually notice that he has not been paid by Alice, who claims she has paid him, but this is not of our concern since it is not a computation problem.

The alternative, Cipher Block Chaining, is slightly more involved and as a result does not have these security risks. As with ECB, the blocks are split up into appropriate sizes. However, before encrypting a block with CBC, it is Exclusive-ORed with the previously encrypted block, thus chaining the block to its predecessor. Decryption works in a similar way to encryption, but is slightly more complicated and the XOR operation occurs after the block is decrypted.

As can be seen, each block is chained to its predecessor, thus the two problems with ECB are not present — a cipher text now not only relies on the plain text and the key, but also the previous block, thus removing the previously discussed problems with ECB. For any given key, any single plain text will not (necessarily) result in the same cipher text. Changes to the message between transmission and reception (e.g. switching blocks as illustrated above) will result in a message being decrypted to gibberish.

Although CBC removes the main problems ECB exhibits, it still has its own flaws, of which some are security related [25] and some are implementation related. An example of an implementation related issue is the extra steps involved in the application, which increases the running time.

Even though ECB was a slightly weaker mode to use, the decision was taken to use it for this investigation for two main reasons. Firstly, it simplified the applications, since there was not any extra work do be done in manipulating the blocks other than the main encryption/decryption.

The second and probably more important reason is related with the aim of this investigation. Since the only concern is investigating the algorithm, the chaining process is not really of interest — the run-time of the algorithm and not the entire encryption/decryption process is the subject of interest. If the applications were to perform CBC instead of EBC, an extra, unnecessary, step to the process which would not improve the investigation in any way.

3.1.4 Key and Block Sizes

Rijndael was designed to work on a variety of key and block sizes, not just those specified in [13]. With minor modifications, Rijndael can support other key and block sizes, as long as they are multiples of 32-bits. These modifications do not change any of the major workings of the algorithm. Instead they only add new cases in the `GetNumberOfRounds` and `ShiftOffset` functions. These additions simply extend the functions for the non-AES instances of `Nb` (Block Size), `Nk` (Key Size) and `Nr` (Number of Rounds). The exact details of this can be found in [6].

Although it would be interesting to observe how Rijndael performs with these alternative key and block lengths, the main use of Rijndael is as the Advanced Encryption Algorithm. As such, these extra lengths are likely to be unused and whilst it would be informative to discover how they perform, it is not as important as considering a block size of 128-bits and key sizes of 128-, 192- and 256-bits.

Also, it has been shown in [6] that the key length influences the run-time of the process — the longer the key, the longer the process. This shows that the key length parameter is unable to be fixed and as such, need to consider the three key sizes need to be considered in separate tests.

3.1.5 Key Management

An area of major importance in cryptography is Key Management — A message is encrypted using a certain key, but how is the key safely sent to the receiver of this message? If using a Public Key cryptosystem (such as RSA [24]), then this is not a problem — the public key (which as the name suggests, is freely available) can only be used to encrypt the message, not to decrypt it. However, since a Symmetric algorithm is being used, this does not apply.

There are many solutions to this (such as using Key-Encryption Keys), but are all outside the scope of this project. As such, each of the applications will use basic (unsecure) methods for passing the key between the encryption and decryption programs. More information regarding the problem of Key Management can be found in [2] and [9].

3.1.6 Padding

As described in section 3.1.4, blocks of 128-bit size will be used for the encryption/decryption. However, what would happen if the input was not a multiple of the block size? The last block would not be of the correct size and so cannot be entered as input for the algorithm. Naturally, to get around this it is necessary to pad out the remaining bits of the final block before encrypting, but this creates a problem when decrypting - how do you know how much of the final block is padded data or real data?

One solution could be as follows:

- Pad the final block with a random series of 0's and 1's (Set randomly so as not to give any information away to cryptanalysts);
- Append an extra block to the input, with the first 120-bits (again set randomly) and the final 8-bits set to store the number of padded bits in the previous block;
- Encrypt this new input as normal.

This solution is suitable for software implementations, but the extra memory required for the appended block may be of concern in Restricted-Space Environments. It is not a perfect solution since there must always be an extra block appended to the end of the input, but it does illustrate that the original problem can be overcome.

However, much like CBC mode, this adds extra complexity to the application, a complexity which is not beneficial to the quality of this investigation. Although this is an important problem in cryptography, it is not the concern of the project, therefore the applications can simply assume the input is of the correct size, i.e. a multiple of 128-bits.

3.2 Functional and Non-Functional Requirements

After analysing the various aspects of Rijndael and cryptography, it is possible to establish the Functional and Non-Functional requirements of the implementations and their applications.

3.2.1 Functional Requirements

For each language considered, two pieces of software need to be developed:

- An encryption program to convert a supplied plain text into an encrypted cipher text, using a randomly generated key;
- A decryption program to convert the cipher text into the original plain text, using the same key that was used for its encryption.

Secure Key Management need not be considered as it is not the focus of this study. It is only necessary to ensure the key used to encrypt the plain text is available for when decrypting the cipher text.

3.2.2 Non-Functional Requirements:

The Rijndael implementations should:

- Be of the traditional ‘algorithmic’ style, rather than tabular;
- Handle 128-bit, 192-bit and 256-bit keys;
- Be modular in implementation so other software can use the developed software.

The applications should:

- Work on both Windows and Unix systems;
- Be appropriate according to the language used;
- Use Rijndael in Electronic Cookbook mode;
- Divide the input into 128-bit blocks;
- Assume the input is an exact multiple of the Block Size;

3.3 Expected Findings

The software use of Rijndael allows for a varying scope of implementations since there are many factors to consider: application, memory constraints and processor speed among others. However, some these factors can also affect the performance of the cryptographic process. For example, the slower the processor, the fewer cycles per second, which results in a lower number of instructions that can be computed per second. As was discussed in the previous section, there are also many factors within the implementation of the software which will affect the performance of the algorithm. These factors include implementation style and cryptographic mode, both of which also influence the performance.

If the above factors are fixed so they do not have an effect on the performance of the algorithm, essentially there is an environment which contains only one variable factor, that of the language used in the implementation. As documented above, the more general purpose and widely used languages have a richer history in Cryptography than the other languages being considered. Therefore, it is to be expected that these languages would run the algorithm quicker than the languages which were designed for a more specialist usage.

As C and Java implementations were used to describe performance figures of Rijndael in [6] then coupled with the general nature of these languages, it is to be expected that these implementations will perform the best. Since Perl and JavaScript are also well designed, multi-purpose languages with a history in cryptography, these will perform well. However, their performance will not be as quick as the C or Java implementations because they are interpreted languages, not compiled, so will not be as optimised as the C or Java implementations. After these languages, it is predicted that the Common Lisp and Matlab implementations will occur, since they are both languages which have their strong aspects, but are not quite as optimised as the other languages being considered.

In section 3.1.4, it was indicated that the experiments would need to be performed for the three standard AES key sizes. Whilst the key size has an impact on the absolute values of each experiment, it should affect each implementation equally and not influence any one implementation over another since the longer the key, the longer the algorithm takes to execute. Also, the experiments are being performed for both encryption and decryption which, like key size, will affect the absolute values of each implementation equally, but not influence one over the other.

In summary, it is expected that the overall performance figures of the implementations will be divided into three distinct groups: the first containing the compiled general purpose languages, C and Java; the second containing the interpreted general purpose languages, Perl and JavaScript; finally the other languages under consideration, Common Lisp and Matlab.

Chapter 4

Implementation

Now that the experiments have been defined and the variable factors of the algorithm and applications fixed, it is time to develop each implementation. The source for the C implementation can be found in Appendix B, whilst all the code is located on the accompanying CD. This chapter is not intended to be an in depth discussion of the software produced for each implementation, merely a detailed description of the *process* undertaken whilst developing each implementation.

4.1 Algorithm

The implementation process of this project is not the same as the standard implementation process for many other software development projects. As the same algorithm is being coded in six different languages, the experience gained in the first development cycle will be used when developing the later implementations.

4.1.1 C Implementation

This first implementation was developed using the specifications supplied in [6] in C. It was implemented as a C module, so that to use the encryption and decryption functions the application simply had to include the Rijndael header file. The functions `encrypt` and `decrypt` were then available to encrypt and decrypt blocks of data.

At this stage, only the algorithm was being developed and a single test vector was used to ensure the algorithm was being developed correctly. This test vector was an input of $\pi \times 2^{124}$ and a key of $e \times 2^{124}$, but was not an arbitrary choice. An alternative specification for Rijndael [10] has been written by Dr Brian Gladman, who also provided three worked examples of show the algorithm working, one of which was the key/input pair mentioned previously.

This document is extremely useful as it provides a step-by-step walkthrough of the encryption process — it describes how the State should appear after each step of the algorithm and can be used as a quick test to make sure the algorithm being created works correctly. Conversely, if the algorithm does not work correctly then it is very useful, as it can show where the error(s) lie in the code very quickly by identifying where the algorithm differs with the specification. Of course, this cannot replace the full testing process (described in detail in Chapter 5), but was extremely useful as it ensured that the implementation being developed was, up to full testing, implemented correctly.

Once the C implementation had been finished and tested using the above test vectors, the other test vectors supplied in [10] (same input, but with keys of $e \times 2^{188}$ and $e \times 2^{252}$) were tested to ensure the software could handle the appropriate key lengths. Unfortunately, this testing flagged up an error which had a major impact on the use of the algorithm.

Initially, the algorithm took two arrays as arguments (the Input block and the Cipher Key) and used the size of these arrays to calculate the block and key size for use in the algorithm. However, when testing the input with the 192-bit key ($e \times 2^{188}$), the encryption algorithm correctly produced the output '`f9fb29aefc384a250340d833b87ebc00`'. When this cipher text is passed to the decryption algorithm, the `length(Input)` function, which returns the length of the array `Input`, incorrectly returned 15 and not 16. This is due to the last element of the array, '00', being another representation of the String Terminating character, '\0'. C assumes this to be the end of the array, when in fact it was an element of the array.

To overcome this, it was unfortunately necessary to pass the lengths of the Input and Cipher Key arrays to both the encryption and decryption functions explicitly. So now, instead of having the function call `encrypt(Input, CipherKey)`, it is necessary to call the function with `encrypt(Input, BlockLength, CipherKey, KeyLength)`.

This is undesirable since it adds extra complexity to the application — the application should not have to pass the block and key sizes to the algorithm explicitly, rather it should be implicit in the lengths of the array. Although it is not possible with the C implementation, it should still be attempted within the other implementations.

Once the C implementation had been created (and shown to work for the three test vectors in [10]), it was necessary to create the other implementations based on this C version. As described in Chapter 3, each implementation was to be as similar as possible. Thus, the process of creating these new implementations is simply converting the C code into the relevant language.

4.1.2 Perl Implementation

The conversion process was the simplest with Perl since, as a language, Perl is very similar to C. The only real changes were syntactic, such as how functions are declared and variable naming. As with C, the three test vectors from [10] were applied and the main problem with the C implementation was not present — the block and key lengths were implicitly passed to the algorithms through the array lengths correctly.

4.1.3 JavaScript Implementation

The only real complication with the JavaScript implementation was with its lack of pointers. Pointers were used throughout the C implementation to alter variables which had been declared outside the current scope (such as the State). This was overcome by passing the variable to the function by using Pass-By-Value, altered as necessary and then returned by the function.

This was not difficult for the current software since only one variable was manipulated inside the function and so returned. If more than one variable was altered, then the multiple altered variables would not be returned so easily. Fortunately, this was not a concern in this project, since this problem was not a factor in the software being developed.

4.1.4 Java Implementation

The Java implementation was slightly more complicated because, although it is syntactically similar to C, it is conceptually different; C is a procedural language, Java is Object-Orientated. Therefore it was necessary to wrap the algorithm inside an Object. The main function for both encryption and decryption has been placed inside the constructor of this Object. This means the instantiation of a new Object is the equivalent 'trigger' for the cryptographic process, much like calling the `encrypt()` or `decrypt()` functions.

Since the constructor returns a new Object and not any other value, it is not possible to return the result of the encryption/decryption process using the same mechanism as the C implementation. To get around this, it has been necessary to add a new function, `getOutput()`, which does return the output of the process.

This is not particularly desirable, although is unavoidable, as the whole process now takes two lines of code, rather than just one:

In C:

```
01 CipherText = encrypt(PlainText, CipherKey);
```

In Java:

```
01 Encrypt e = new Encrypt(PlaintText, CipherText);
02 CipherText = e.getOutput();
```

4.1.5 Common Lisp Implementation

The Common Lisp implementation was attempted before Matlab on the grounds that the author was more familiar with Lisp (although not Common Lisp), than Matlab. [29] was particularly useful for explaining the differences between EULisp (the dialect of Lisp the author was familiar with) and Common Lisp.

The conversion process took much longer for Common Lisp than the previous transformations because of the major differences in syntax. However, once the initial language difficulties had been overcome, there were not any major problems other than in converting the S-Boxes from hexadecimal into denary.

4.1.6 Matlab Implementation

Matlab was a new language to the author and so needed to be learnt first. Although this was not particularly difficult, [11] was helpful. Once the different style of implementation had been fully grasped (with each function needed for both encryption and decryption having to be stored in separate files), it was just a case of converting the syntax.

4.2 Applications

As described in Chapter 3, the applications which make use of each implementation are very basic. However, they are still an important part of this investigation as they demonstrate the use of Rijndael in each implementation.

As the major testing was being not being performed at this stage, the test vectors in [10] were used to test each of the applications, as well as the obvious axiom that the decryption of the cipher text should be the same as the original plain text.

The C and Perl applications are very similar since they both deal with encrypting and decrypting files. Much like the development of the algorithms, the C file was developed initially and was then converted into the Perl application, ensuring the different file modes (writing binary data in C, and text in Perl) were adhered to.

The JavaScript application was the simplest application to create since most of the work is handled by the web browser. The PHP and HMTL code used to receive the user input and display the result of the cryptographic process was trivial. The main problem with this application was ensuring it worked in the different browsers, such as Opera and Microsoft Internet Explorer. The complications were not caused by the algorithm, but rather the syntax used in the application to display information about the encryption/decryption process (such as the Input array, Cipher Key and final Output array).

Unlike the JavaScript application, the Java application was the most complicated to develop. The problem did not lie in the underlying functionality, but with developing the graphical interface for the applet, the Layout Managers in particular. Although the user interface is not of concern, the applet should still be useable, so it was still necessary to spend the time in getting this correct.

Chapter 5

Testing

The testing phase is an important part of software development since it helps to ensure that the produced software conforms to the Requirements Specification. It should be noted from the above description that testing does not ensure the developed code works as intended, since there may be flaws in the Requirements Specification which are then replicated in the code.

For this project however, this is not a problem — a series of Test Vectors exists with which it is possible to compare the implementations against. These vectors were produced by the developers as part of the AES Submission and come in two types — one which fixed keys for a standard input text, the other for fixed text and varying keys.

For both of these types, there are three groups of tests — one for 128-bit keys, one for 192-bit keys and the other for 256-bit keys. The input text is always fixed at 128-bit. The tests are set up as described in the tables below.

As can be seen, 960 test files will be considered for each implementation, resulting

128-bit Keys		
Fixed Key Tests	Keys	Fixed at 0000000000000000
	Text	128 different tests - each corresponding to a field of 0s, with a different bit set accordingly (e.g., Test 1 would be 100...000, Test 2, 010...000, Test 128, 000...001, etc.)
Fixed Text Tests	Keys	128 different tests - each corresponding to a field of 0s, with a different bit set accordingly (e.g., Test 1 would be 100...000, Test 2, 010...000, Test 128, 000...001, etc.)
	Text	Fixed at 0000000000000000

Table 5.1: 128-bit Key Test Vectors

192-bit Keys		
Fixed Key Tests	Keys:	Fixed at 000000000000000000000000
	Text:	192 different tests - each corresponding to a field of 0s, with a different bit set accordingly (e.g., Test 1 would be 100…000, Test 2, 010…000, Test 192, 000…001, etc.)
Fixed Text Tests	Keys:	128 different tests - each corresponding to a field of 0s, with a different bit set accordingly (e.g., Test 1 would be 100…000, Test 2, 010…000, Test 128, 000…001, etc.)
	Text:	Fixed at 0000000000000000

Table 5.2: 192-bit Key Test Vectors

256-bit Keys		
Fixed Key Tests	Keys:	Fixed at 00000000000000000000000000000000
	Text:	256 different tests - each corresponding to a field of 0s, with a different bit set accordingly (e.g., Test 1 would be 100…000, Test 2, 010…000, Test 256, 000…001, etc.)
Fixed Text Tests	Keys:	128 different tests - each corresponding to a field of 0s, with a different bit set accordingly (e.g., Test 1 would be 100…000, Test 2, 010…000, Test 128, 000…001, etc.)
	Text:	Fixed at 0000000000000000

Table 5.3: 256-bit Key Test Vectors

in 11,520 tests overall — encrypting each of the 960 test files, then decrypting the output for each of the 6 implementations. Given that the Java, JavaScript, Matlab and Common Lisp implementations require user input, it was decided to test the encryption and decryption algorithms using scripts. Unfortunately, this would require drastic changes to the original applications, although due to modularisation, the algorithm did not need to be changed.

These applications[1] were created in a similar style to that of the Perl application (converting text files) and were essentially converted from Perl into the respective languages. The original C and Perl applications also needed to be amended to accommodate the utilisation specific keys — in the demonstrative applications, the key was generated randomly at run-time.

Once these changes had been made, batch files were then created to run all 11,520 tests. Once these test were run, the individual output files were merged into vari-

[1]The source code for which, along with compiled binaries where necessary, are available on the attached CD in the *Testing* folder.

ous files for comparison later on. The mergings were based only on implementation language and variable component. For example, a single file contain all the output regarding the encryption and decryption of the C implementation for varying keys. These output files were then compared against the original test vectors. Tests were flagged up which had incorrect Cipher Texts, or Decrypted Texts which were different from the original Plain Text.

The testing showed that the majority of the implementations worked correctly. However, there were two main issues that the original testing highlighted, concerning the Perl and Matlab implementations. The Matlab bug was due to a typo in copying the S-Box and Inverse S-Box (from a previous implementation), and once this was changed and re-tested, everything operated as it should. However, this caused the need to check the other five implementations' S-Boxes to ensure they were correct, which was painstaking work, but a necessary verification.

The issue with the Perl implementation was its handling of the newline character in Windows. Since the Perl implementation was dealing with characters, if after encryption, one of the encrypted bytes was a newline character ('\n', or '`0A`' in hexadecimal ASCII), the problems would arise when writing to the output file. Since characters (and not individual bits) were written to file, Windows would convert any newline characters into its equivalent when saving - a Carriage Return and a Line Feed ('\r\n', or '`0D0A`' in ASCII). Thus, when the output file was being decrypted, it would find 136-bits to decrypt and not the expected 128-bits, resulting in an incorrect decryption.

To overcome this, it was necessary to tell Perl that if it discovered '\r\n' then it should actually consider it as being the '\n' character. This may lead to some confusion as to what would happen if in the cipher text, a '\r' preceded a '\n', but none would occur as it was explicitly told to treat these as two separate characters during saving.

Once these changes had been made, it was imperative to run the appropriate tests again to see if the changes resulted in valid programs. This time through, there were not any problems; all the appropriate tests were re-run and all passed without a single failure.

However, this was not the end of the testing. All that the above had shown was that the algorithms for encrypting and decrypting blocks of data worked appropriately. It did not show if the applications were correct. Since the applications were not the main concern of the project, a less strict test plan was used, whereby only a handful of test cases were utilised (including some which were known to cause issues).

Compared to the previous testing, this was more straightforward, since it was known that the underlying code was correct. Therefore, it was just testing whether the applications handled the data suitably. Due to the simplicity of these applications and the testing performed during their development, all these tests passed successfully.

The evidence of the testing, including the test application output, merged results and comparison files are available on the attached CD in the *Testing/Results* folder.

Chapter 6

Experimental Results

Once the algorithms had been tested, it was imperative to perform the experiments that were designed in Chapter 3. The final aspect that needed to be defined for the experimental design to be completed was the sample size. This was not determined in Chapter 3 because it was decided that this value would be partially dependent on the implementations. As it turned out, this was not the case, but it still had to be defined.

To create a large enough sample size for our experiments, it was decided that 128 tests should be performed for each experiment. This is a sufficiently large enough number to enable the identification of any trends, but is not so large as to over-complicate the analysis of the data. Also, to create a more accurate estimate of the mean (by removing the effect of outliers), each of the 128 experimental results will be derived as the mean of 128 runs of the algorithm. The number 128 was chosen entirely arbitrarily, and its only real relevance is that it is 2^7!

6.1 Application Creation

To carry out the experiments, it was impractical to use the applications created to demonstrate the languages. Due to the large sample size it would be infeasible to manually enter the data in some of the implementations (such as the Java application). As such, it was necessary to create new applications which would record the timings of each run.

Unfortunately, it was not possible to simply extend the applications created for Chapter 5 to record the run times of each implementation. The reasoning behind this was that the run-time of a single execution of the algorithm is very fast, mostly less than a second[1], It was not possible to simply use the computers clock to calculate the run-time since millisecond precision was required.

[1]The exception being decryption in Matlab.

To overcome this, specialist functions (such as the `clock()` function in C) were needed. Unfortunately, these functions were not precise enough for the C implementation (the entire `encrypt()` function always ran in a total of 0 clock ticks!), therefore meaning an alternative solution had to be found.

The chosen solution was to discard the applications used in testing and create a new application. This was to be a simple application, which would calculate the overall run-times of performing the algorithm 128 times to calculate the overall mean for this execution. A single input and key would be used because now the concern is not with the output of the algorithm, but the time it takes to compute the output.

The source code for the applications used in the experiments, along with compiled binaries where necessary, are available on the attached CD.

6.2 Results

The majority of the applications were executed on a 600MHz Intel Mobile Celeron™ processor, running Microsoft® Windows® Millennium Edition machine, running minimal applications. Unfortunately, the Matlab implementation had to be run on a Sun Ultra E3500 Solaris machine on the University of Bath campus network (named ‘midge’ and has 8 $\times$ 400 MHz UltraSparc II CPUs) to make use of the University’s site licence.

The C implementation was compiled using [20] and so was run as its own application. The Java implementation was compiled using [30], but was executed using the interpreter that is available with the compiler. The JavaScript implementation was executed using the Opera 8.0 browser whilst the Perl version was run using the interpreter [1]. Finally, both the Common Lisp and Matlab implementations were run in their respective interpreting programs.

The output that each application generated was then concatenated into a single file, resulting in 36 output files — one for each language implementation, for each test. These output files were then inserted into a spreadsheet and used as the base for the statistical analysis performed in Chapter 7.

For brevity, the following is just a summary of the results obtained. The full results can be found on the attached CD in the *Results* folder.

6.2.1 Encryption with 128-bit Keys

Language	*Mean Time*
C	0.000433350
Java	0.000444946
JavaScript	0.022249756
Common Lisp	0.068173826
Perl	0.019009399
Matlab	0.412536165

Table 6.1: Encryption with 128-bit Keys Test Results

6.2.2 Encryption with 192-bit Keys

Language	*Mean Time*
C	0.000437662
Java	0.000440674
JavaScript	0.027895508
Common Lisp	0.087386475
Perl	0.023798828
Matlab	0.442708932

Table 6.2: Encryption with 192-bit Keys Test Results

6.2.3 Encryption with 256-bit Keys

Language	*Mean Time*
C	0.000435181
Java	0.000429077
JavaScript	0.032866821
Common Lisp	0.101591797
Perl	0.027548218
Matlab	0.627511749

Table 6.3: Encryption with 256-bit Keys Test Results

6.2.4 Decryption with 128-bit Keys

Language	*Mean Time*
C	0.001325684
Java	0.000701294
JavaScript	0.092683716
Common Lisp	0.297310176
Perl	0.074577026
Matlab	2.153083282

Table 6.4: Decryption with 128-bit Keys Test Results

6.2.5 Decryption with 192-bit Keys

Language	*Mean Time*
C	0.001559448
Java	0.000794678
JavaScript	0.112312012
Common Lisp	0.361439818
Perl	0.089953613
Matlab	2.143793864

Table 6.5: Decryption with 192-bit Keys Test Results

6.2.6 Decryption with 256-bit Keys

Language	*Mean Time*
C	0.001905518
Java	0.000926514
JavaScript	0.130484619
Common Lisp	0.420760496
Perl	0.104390259
Matlab	3.054164864

Table 6.6: Decryption with 256-bit Keys Test Results

Chapter 7

Analysis of Results

After collecting the data, it is now essential to analyse the results using appropriate statistical methods. This is required because it is not possible to come to any sensible conclusions using only the mean of the data — it is necessary to perform some form of statistical analysis on the data. Due to the nature of the comparison, the orderings of these groups can be easily determined without using statistical analysis, and this will be discussed further in section 7.3. The start of this Chapter, therefore, is a discussion of the Analysis performed in trying to discover groupings in the data.

Also, it would be infeasible to include all the analysis in this section due to the number of experiments being performed. As such, the following will be a description of the methods used in analysing the data, using the data generated from the Encryption with the 128-bit Key tests as an example. Appendix A contains the results from the rest of the analysis.

As large quantities of data has been generated by the experiments in Chapter 6, only the calculated means for each implementation will be quoted in this write-up. The full data set can be found for each experiment in the spreadsheets in the *Analysis* folder on the attached CD.

7.1 Data Set

The following data was generated from the Encryption with 128-bit Key tests, and will be used in the proceeding sections to illustrate the various methods.

Language	*Mean Time* ($\hat{\mu}$)
C	0.000433350
Java	0.000444946
JavaScript	0.022249756
Common Lisp	0.068173826
Perl	0.019009399
Matlab	0.412536165

Table 7.1: Data Set generated from Encryption with 128-bit Keys

7.2 Analysis of Variance

An Analysis of Variance tests whether differences exist between the mean times, but does not pinpoint exactly where in the data the differences occur. Therefore, the aim of Analysis of Variance is to test whether differences actually exist before a more in depth statistical analysis is performed on the data to locate any differences shown.

7.2.1 Hypothesis

The experiments are performed under the assumption that the data follows a Normal distribution [39] with the properties

$$\mathrm{Y}_{ij} \sim \mathrm{N}(\mu_j, \sigma^2)$$

where $i = 1, \ldots, 128$ (Tests) and $j = 1, \ldots, 6$ (Languages) are independent results — regardless of language, the result of one test is not influenced by the result of a previous test.

To perform the tests, first it is necessary to define our Null Hypothesis, H_0, along with an alternate hypothesis, H_1:

H_0: $\mu_1 = \mu_2 = \ldots = \mu_c$.
H_1: At least two μ_{j_h}, μ_{j_t} ($j_h \neq j_t$) are not equal.

where μ_j is the mean of language j in the data set.

The hypothesis H_0 states that the column means are the same, whilst the alternate hypothesis claims this is not the case. In other words, H_0 states that the run-times of each program are the same and thus it would be unnecessary to perform any further analysis.

Using the example data:

H_0: $\mu_C = \mu_{Java} = \mu_{JavaScript} = \mu_{Perl} = \mu_{Matlab} = \mu_{Common\ Lisp}$
for the Encryption with 128-bit Key test
H_1: At least two are different.

The μ used in the statement of the hypothesis is the theoretical mean for each language. The theoretical mean is obtained from taking the average of the population. With out data it is not possible to calculate the theoretical mean because the programs can be run a countably infinite number of times. Instead, an estimate $\hat{\mu}$ for μ is calculated based on the sample taken and used in the analysis.

7.2.2 Analysis of Variance Table

Now that the background information for the analysis has been detailed, it is necessary to define the method this analysis will use to determine if there is sufficient evidence to reject H_0 or not. This starts by constructing an Analysis of Variance Table, which calculates certain properties of the data set, which will be used later come to conclusions on the data. This table is constructed as below:

Source	Sum of Squares	df	Mean Square
Column Sum of Squares(CSS)	$\frac{1}{r}\sum_{j=1}^{c}\left(\sum_{i=1}^{r} Y_{ij}\right)^2 - \frac{1}{rc}\left(\sum_{i=1}^{r}\sum_{j=1}^{c} Y_{ij}\right)^2$	c - 1	$\frac{CSS}{df}$ = CMS
Residual Sum of Squares(RSS)	RSS = TSS - CSS	$c(r-1)$	$\frac{RSS}{df}$ = RMS
Total Sum of Squares (TSS)	$\sum_{i=1}^{r}\sum_{j=1}^{c} Y_{ij}^2 - \frac{1}{rc}\left(\sum_{i=1}^{r}\sum_{j=1}^{c} Y_{ij}\right)^2$	$cr-1$	

Table 7.2: Analysis of Variance Table

In Table 7.2, df is the Degrees of Freedom (the number of independent parameters), TSS is the total variation, CSS is the variation between groups and RSS is the variation within groups.

Source	Sum of Squares	df	Mean Square
Column Sum of Squares(CSS)	16.65650918	5	3.331301836
Residual Sum of Squares(RSS)	0.022956638	762	3.01268×10^{-5}
Total Sum of Squares (TSS)	16.67946582	767	

Table 7.3: Analysis of Variance Table for Encryption with 128-bit Keys

Applying the formulae in Table 7.2 to the example data, the result is the following Table 7.3. These values will then be used at a later stage to test the null hypothesis.

7.2.3 Test Statistic

A Test Statistic, $\tilde{F}$ is defined, which is used later to either reject H_0 or show there is insufficient evidence to reject it.

$\tilde{F}$ is calculated as:

$$\tilde{F} = \frac{\left(\frac{CSS}{df}\right)}{\left(\frac{RSS}{df}\right)} = \frac{CMS}{RMS}$$

If H_0 is true, $\tilde{F}$ follows an F distribution [38]

$$F_{Column\ df,\ Residual\ df} = F_{c-1,c(r-1)} = F_{5,762}.$$

i.e. Under H_0: $\tilde{F} \sim F_{5,762}$

If $\tilde{F}$ is too large for an $F_{5,762}$ distribution, H_0 is rejected.

Using statistical package R [32], it is possible to find an appropriate F value according to level of significance required. The standard level of significance is the 5% level, which for our analysis produces $F_{5,762,0.05} = 2.225856$. So, if $\tilde{F} > 2.225856$ there is significant evidence to reject H_0: there exist differences between each of the run-time means. If $\tilde{F} < 2.225856$ there is insufficient evidence at the 5% significance level to reject H_0: each column run-time is the same.

The $\tilde{F}$ value for the example data is calculated as 110575.9468. This value is clearly greater than the value from the F distribution, indicating there is sufficient evidence to reject H_0. This tells us that differences exist within the calculated means of the data.

Now it has been established through Analysis of Variance that differences exist, it is now necessary to identify where these differences lie, i.e. between which of the means. There are two tests to do this:

- Least Significance Difference
- Bonferroni Test

7.2.4 Least Significance Difference

A Least Significance Difference test uses the following hypothesis, as well as the proceeding condition to check if any data pair is significantly different or not.

H_0: $\mu_j = \mu_k$	$j \neq k$	Means of languages j and k are the same.
H_1: $\mu_j \neq \mu_k$	$j \neq k$	Means of languages j and k are different.

For a test of significance level α, if | $\hat{\mu}_j$ - $\hat{\mu}_k$ | $>$ $\sqrt{\frac{2s^2}{r}}$ t $_{c(r-1),\frac{\alpha}{2}}$ reject H$_0$: there exists significant differences in means of languages j and k. t$_{c(r-1),\frac{\alpha}{2}}$ is the corresponding t-Distribution [40].

In the example, the condition is:

| $\hat{\mu}_j$ - $\hat{\mu}_k$ | > 0.001346868

7.2.5 Bonferroni Test

A Bonferroni Test uses the following hypothesis, as well as the proceeding condition to check if any data pair is significantly different or not.

H$_0$: $\mu_j = \mu_k$	$j \neq k$	Means of languages j and k are the same.
H$_1$: $\mu_j \neq \mu_k$	$j \neq k$	Means of languages j and k are different.

This time, H$_0$ is rejected at the significance level α, if | $\hat{\mu}_j$ - $\hat{\mu}_k$ | $>$ $\sqrt{\frac{2s^2}{r}}$ t $_{c(r-1),\frac{\alpha}{2m}}$, where m is the number of tests and t$_{c(r-1),\frac{\alpha}{2m}}$ is the corresponding t-Distribution [40].

For the example, the condition is:

| $\hat{\mu}_j$ - $\hat{\mu}_k$ | > 0.002015952

7.2.6 Analysing Output

Least Significance Difference leads to too many significant results, where as Bonferroni Tests lead to too few. The intersection of these two results shows a definite significant difference, whilst a significance flagged by Least Significance Differences but not Bonferroni Tests means that there may be a significant difference, but it is not possible to be sure.

Table 7.4 was the resulting output for applying both the Least Significance Difference and Bonferroni tests on the example data. In the table, the LSD column is the result of the Least Significance Difference test, and the BT column is the result of the Bonferroni Test. For both these columns, a 'False' result indicates that there was insufficient evidence to reject H$_0$ (i.e. $\hat{\mu}_j$ = $\hat{\mu}_k$), whilst a 'True' result indicates that there was sufficient evidence to reject H$_0$ (i.e. $\hat{\mu}_j \neq \hat{\mu}_k$).

7.2.7 Test Results

The tables generated by the Least Significance Difference and Bonferroni Tests, such as Table 7.4, are very useful at quickly indicating whether there is sufficient evidence or not to reject the null hypothesis, H$_0$. However, it is the status of H$_0$ which is the most useful when drawing conclusions from the raw data.

Lang j	Lang k	$\lvert\hat{\mu}_j$ - $\hat{\mu}_k\rvert$	LSD	BT
C	Java	1.15967×10^{-5}	False	False
C	JavaScript	0.021816406	True	True
C	Common Lisp	0.067740476	True	True
C	Perl	0.018576050	True	True
C	Matlab	0.412102815	True	True
Java	JavaScript	0.021804810	True	True
Java	Common Lisp	0.067728879	True	True
Java	Perl	0.018564453	True	True
Java	Matlab	0.412091219	True	True
JavaScript	Common Lisp	0.045924070	True	True
JavaScript	Perl	0.003240356	True	True
JavaScript	Matlab	0.390286409	True	True
Common Lisp	Perl	0.491644260	True	True
Common Lisp	Matlab	0.344362339	True	True
Perl	Matlab	0.393526765	True	True

Table 7.4: Output of Least Significance Difference and Bonferroni Tests

In the example for Encryption with 128-bit keys, both the Least Significance Difference and Bonferroni Tests show there is insufficient evidence to reject H_0 when testing for differences between C and Java run-times. However, there is sufficient evidence to reject H_0 for any other combination of languages. This implies that for Encryption with 128-bit keys, the original hypothesis in Chapter 3 was only partially correct — C and Java are similar, but there are differences in run-times between the remaining languages.

7.3 Conclusions

The overall results, shown in Tables 7.1 and A.1-A.5, show that the first part of our hypothesis from section 3.3 was correct — each test produced the expected ordering: C and Java, Perl and JavaScript and finally Common Lisp and Matlab.

There were, however, some surprising results from this, mainly how much slower the Matlab implementation was than the other implementations. This was unexpected because numerical manipulation (and especially those in vectors or matrices) are its forté. A possible reason for this is that the algorithm uses bitwise functions (the Exclusive OR operation in particular), which may not have been implemented for speed in Matlab.

Another interesting result was how much difference there was between the interpreted languages and the compiled (and optimised) implementations. It was expected that there would be a difference between these two types of application, yet this test shows how much of a difference there are between compiled implementations and the uncompiled, interpreted implementations. Reasons for this could be down to the level of optimisation in the optimised code, the language designs for the compiled and interpreted languages used or, more likely, the overheads of running an interpreted language.

The final noticeable result was the difference between the Java and C implementations when decrypting (regardless of key size). The Java and C versions are virtually identical for encryption and although both are considerably slower when decrypting, the C implementation seems to have fared a lot worse than Java. The reasons for this are not precisely known, but by considering the way the algorithm works, it probably lies in the Key Expansion. This is due to the similarities between the rest of the algorithm for encryption and decryption — the only place where these two algorithms differ is the Key Expansion.

The results also illustrated what had already been shown in literature (such as the performance figures in [6]), that the key size also plays a varying part in the performance of the algorithm, but that these changes are uniform across implementations and do not only apply in certain languages.

Now that the individual languages have been analysed, did they group as expected? Unfortunately, the means could not simply be examined to see how similar the languages are, but instead statistic analysis had to be performed, as above.

This analysis returned mixed results — for all six experiments it was possible to show (with a 5% level of confidence) that only the C and Java implementations could be considered similar. This result is unsurprising when considering just how close the calculated means are for encryption. However, when looking at decryption, even though the Java implementation is significantly quicker, it is still very similar as the run-times are in milliseconds.

This fact is also possibly the reason why the Perl and JavaScript implementations cannot be shown to be similar — although the Perl and JavaScript implementations are similar to tenths of milliseconds, this difference is enough to be considered statistically different, even though for all intents and purposes they are different. Another reason for this is down to the definition of the statistical tests used — the statistical analysis tests for equality, whilst the investigation is more concerned with similarity.

The Matlab and Common Lisp implementation can clearly not be shown to be similar at all — either by comparing the means or by statistical analysis. Whilst comparing the other pairs of languages, the differences are negligible, the variations between Matlab and Common Lisp are sevenfold for both encryption and decryption which, given the relatively slow run-time for Common Lisp, is quite drastic. As discussed earlier, this difference is probably due to the use of the non-standard bitwise functions in Matlab, which is possibly not meant to be used as extensively as it is in Rijndael. This is not so much of a problem in Common Lisp, as it is designed as a more general purpose language and thus has well designed bitwise functions.

Chapter 8

Conclusions

The aim of this project was to investigate the effect of programming languages in developing software implementations for the encryption algorithm Rijndael. Rijndael was selected as the Advanced Encryption Standard to replace the ageing Data Encryption Standard, which had been the previous standard for protecting non-classified US government data.

Initially literature was examined to discover more about the history of the new standard, as well as how the languages chosen had been previously used in the field of cryptography. The information gathered from this literature then helped in deciding how important factors of the algorithm and cryptography would be captured in the experiments.

These factors ranged from the trivial (such as Key and Block sizes), to the slightly more important, (Cryptographic Mode) and finally the fundamentally significant factors (such as Key Management and Padding). Several aspects were taken into account when deciding how to handle these factors, mainly that of need and quality. Only those factors that were necessary and would be of value to the investigation would be considered.

Certain factors, like as Key Management, which have already been identified as very important in cryptography, have been completely ignored in this project, since they are not the focus of the investigation. These factors have a profound impact on cryptography (symmetric algorithms are only as safe as the protection of the Cipher Key). However, they do not have much importance in this investigation and so can safely be ignored.

Once the experiments had been designed, it was time to implement the algorithm in each language, followed by the demonstration applications. This process was the most time consuming, although after the first implementation had been completed, each remaining implementation was essentially a syntactic transformation.

There were various challenging bugs in the implementation, the most notable of which was correctly calculating array sizes in C. These were flagged whilst doing basic testing, using a small set of test vectors. Once these basic bugs (such as typos and logic errors) had been removed, the full set of test vectors (supplied by the designers) was used for a more thorough testing of the algorithms. To accelerate the testing, basic scripts were created to automate the entire testing procedure. This greatly reduced the time needed to perform all the tests as it was not necessary to manually enter all the test vectors for some implementations (such as Java or JavaScript).

This testing only highlighted two major bugs, one of which had serious repercussions for the Perl implementation and how it handled newline characters. This led to a few unwanted changes in the Perl application, but eventually the bugs were eliminated and the algorithm testing could be considered complete. Some minor testing of the applications remained, but was not nearly as thorough since the applications were relatively small programs and the algorithm was the main focus of the study.[1]

Now that the applications had been shown to be correct for the given test vectors, it was possible to proceed with the experiments. Due to certain design restrictions, it was not possible to use either the demonstration or testing applications, so new applications needed to be written. Once these applications had been created, the experiments were run and the data accumulated.

This data was then statistically analysed using Analysis of Variance, Least Significance Difference and Bonferroni Tests, from which it was concluded that there was a wide range of variation in the implementations, some of which was anticipated, others of which were. For instance, as was expected, the C and Java implementations performed significantly quicker than the other implementations and could be considered to be statistically similar. Conversely, the Common Lisp and Matlab implementations were not as similar as expected and in fact, the Matlab implementation was significantly slower than Common Lisp.

It was also evident that Rijndael applied to interpreted languages is significantly slower than compiled languages. A number of reasons, including: interpreting overheads, code optimisation and language design could explain this.

Finally, as noted in section 6.2, the Matlab experiments were performed on a different machine than the remaining experiments. The reason for this was due to the availability of the software — the Matlab environment was only available from the University server. Unfortunately, it was not possible to perform all the experiments on the University server since the JavaScript implementation has a Client-Side application and is run locally on the users machine.

As it was not possible to perform all the experiments on one machine, it was de-

[1]It should be noted that whilst the applications were tested *after* the algorithms, the bug regarding newlines in Perl was flagged by one of the test vectors and so the relevant changes could be made to the Perl application before the main application testing.

cided to perform the Matlab tests on the University server and the remaining tests on the author's machine. This does result in skewed data, the impact for which cannot be identified exactly. However, C and Perl tests were performed and resulted in similar enough run times that this was not considered a problem. Of course, this definition of "similar enough" may have an impact on the results, but was not considered serious enough.

Unfortunately, this investigation was restricted to a certain period of time and as such, there are still many more experiments that could be of interest using these implementations. These are given in the following section.

8.1 Future Work

At the culmination of this project, there are still plenty of experiments which are a continuation on that which has already been completed. In no particular order:

- **Lower Level Experiments** — It could be seen from the conclusions drawn in section 7.3, there are many unknowns as to why there were such drastic differences in the run-time of each implementation. Therefore, it would have been interesting to discover if this was due to not only particular part(s) of the algorithm being considerably slower, but if so, *which* part(s). This can certainly be applied to the Matlab implementation as it would be interesting to see why it is considerably slower than the other implementations.

  Although this was briefly attempted for the project, it was quickly ceased, mainly for the same reasons that were specified in section 6.1 — that without drastic re-writing of code, it would have be infeasible in the remaining timeframe to perform such analysis. However, it is suspected that if this was due to a particular section of the code, it would be the Key Expansion, especially with decryption.

  From what has been demonstrated in this investigation, this analysis could be performed on the Matlab implementation. since it appears to extenuate the sections of Rijndael (although this assumes Matlab's slow run-times are due to a particular section).

- **Non-AES Key/Block Sizes** — This project concentrated on only the Key and Block sizes specified in the AES since they are the most common sizes when using Rijndael. However, as indicated in Section 3.1.4, Rijndael is fully capable of using a select range of non-standard Key and Block sizes. Therefore it would be interesting to investigate how Rijndael would perform when using these non-AES Key and Block sizes.

- **Encrypting in CBC Mode** — CBC mode does not really affect the run-time of the algorithm (which was the focus of this study), but does have an effect on the run-time of the entire encryption/decryption process. Thus, it would be interesting to examine the impact CBC mode would have on the run-time of a Rijndael-based cryptographic system, compared to that of ECB.

As can be seen, this list of future work could very easily turn into a list of “What are the implications of a different Design Choice is made in section 3.1?”. Naturally, this should be the case, but it is felt that the first two extensions listed above are the most intriguing since it would be interesting to know *where* the algorithm seems to be performing slowly and what the performance figures for non-standard Key and Block sizes would be.

8.2 Personal Comments on the Project

Unfortunately, there have been some design choices which were made at an early stage in the project that now seem a little confusing. The first of these was the purpose of the demonstrative applications. They are not used in the project (other than to demonstrate each implementation), and so seem to have no bearing on the project. Initially, it was planned that these applications were going to perform the testing and the experiments, but due to the size of these tests and the manual input nature of certain applications, they were phased out. This means that the demonstrative applications were probably not a wise choice since they contribute very little to the project.

The other choice which now appears to be a mistake was the use of the statistical analysis method. Although the selected method works well, it is not quite the style of test that was required — the aim was to test for similarity, not equivalence. It is felt that had an alternative method been applied to the data (one which tested for similarity and not equivalence), then, in all likelihood, the Perl and JavaScript implementations would have been considered similar.

Overall, this project was very appealing and was a good choice as it was relevant to the authors interests. It was a challenging investigation as it was not the ‘typical’ type of project the author would have initially selected.

The project was originally selected to be a development assignment, but after contemplating the problem in-depth, it evolved into the current problem — one whose final result was not so obvious. As such more was learnt from this project than would otherwise have been the case.

Bibliography

[1] ActiveState. ActivePerl. http://www.activestate.com/Products/ActivePerl.

[2] D Baleson. Automated Distribution of Cryptographic Keys using the Financial Institution Key Management Standard. *IEEE Communications Magazine*, 23(9):41–46, September 1985.

[3] E Biham. A note on comparing the AES candidates. In *The Second AES Candidate Conference*, The National Institute of Standards and Technology, Gaithersburg, MD, March 1999. The National Institute of Standards and Technology, Gaithersburg, MD.

[4] D Bishop. *Introduction to Cryptography with Java Applets*. Jones and Bartlett Publishers, Inc., 2002.

[5] J Daeman, L Knudsen, and V Rijmen. The Block Cipher SQUARE. http://www.esat.kuleuven.ac.be/∼cosicart/pdf/VR-9700.PDF.

[6] J Daemen and V Rijmen. AES Proposal: Rijndael. http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael.pdf.

[7] E Danielyan. Goodbye DES, Welcome AES. *The Internet Protocol Journal*, 4(2):15–21, June 2001.

[8] N Froyd. Common Lisp Rijndael Implementation. http://www.cs.rice.edu/∼froydnj/lisp/aes.lisp.

[9] W Fumy and P Landrock. Principles of Key Management. *IEEE Journal*, 11(5):785–793, June 1993.

[10] B Gladman. A Specification for Rijndael, the AES Algorithm. http://fp.gladman.plus.com/cryptography_technology/rijndael/aes.spec.311.pdf.

[11] D Hanselman. *Mastering MATLAB 7*. Prentice Hall, Upper Saddle River, N.J. USA, International edition, 2005.

[12] R Lidl and H Niederreiter. *Introduction to Finite Fields and their applications*. Cambridge University Press, 1986.

[13] National Instiue of Standards and Technology. Announcing request for candidate algorithm nominations for the Advanced Encryption Standard (AES). *Federal Register*, 62(177), September 1997.

[14] National Instiue of Standards and Technology. *Data Encryption Standard (DES)*. National Institute of Standards and Technology, October 1999. FIPS PUB 46-3.

[15] National Institute of Standards and Technology. Advanced Encryption Standard (AES) Fact Sheet. http://csrc.nist.gov/CryptoToolkit/aes/aesfact.html.

[16] National Institute of Standards and Technology. NIST website for AES. http://www.nist.gov/aes.

[17] National Institute of Standards and Technology. Announcing development of a Federal Information Processing Standard for Advanced Encryption Standard (AES). *Federal Register*, 62(1), January 1997.

[18] J Nechvatal, E Barker, L Bassham, W Burr, M Dworkin, J Foti, and E Roback. Report on the Development of the Advanced Encryption Standard (AES). *Journal of Research of the National Institute of Standards and Technology*, 106(3):511–576, May-June 2001.

[19] J Nechvatal, E Barker, D Dodson, M Dworkin, J Foti, and E Roback. Status Report on the First Round of the Development of the Advanced Encryption Standard. *Journal of Research of the National Institute of Standards and Technology*, 104(5):435–459, September-October 1999.

[20] C Peters, J-J van der Heijden, and M Khan et al. MinGW - Minimalist GNU for Windows. http://www.mingw.org.

[21] V Rijmen. Rijndael homepage. http://www.esat.kuleuven.be/∼rijmen/rijndael.

[22] V Rijmen. AES: The Selection Process. In *Seminar Rijndael*. Universität Paderborn, April 2002.

[23] V Rijmen. Design Philosophy of Rijndael. In *Seminar Rijndael*. Universität Paderborn, April 2002.

[24] R Rivest, A Shamir, and L Adleman. A method for obtaining Digital Signatures and Public Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[25] B Schneier. *Applied Cryptography*. John Wiley & Sons, Inc, New York, NY, USA, 2nd edition, 1996.

[26] R Sevilla. Crypt-Rijndael-0.05 perl module. http://search.cpan.org/ dido/Crypt-Rijndael-0.05/Rijndael.pm.

[27] M Smid. AES: A Crypto Algorithm for the Twenty-First century. In *Fast Software Encryption Workshop*, March 1998.

[28] M Smid and D Branstad. The Data Encryption Standard: Past and Future. *Proceedings of the IEEE*, 76(5), May 1988.

[29] G Steele. *Common Lisp: The Language*. Digital Press, 2nd edition, 1990.

[30] Sun Microsystems. Java 1.5.0 SDK. http://java.sun.com.

[31] Sun Microsystems. Java Cryptography Extension. http://java.sun.com/products/jce/index.jsp.

[32] The R Foundation for Statistical Computing. The R Project for Statistical Computing. http://www.r-project.org/.

[33] W Trappe and L Washington. *Introduction to Cryptography: with Coding Theory*. Prentice Hall, 2002.

[34] Various. CL^{iki} — The Common Lisp Wiki. http://www.cliki.net/Cryptography.

[35] J Vestgården. Common Lisp Rijndael Implementation. http://folk.uio.no/jornv/aes/aes32.lisp.

[36] J Walker. JavaScrypt. http://www.fourmilab.ch/javascrypt/.

[37] J Weiss. *Java Cryptography Extensions: Practical Guide for Programmers*. Morgan Kaufmann, 2004.

[38] E Weisstein. “F-Distribution.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/F-Distribution.html.

[39] E Weisstein. “Normal Distribution.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/NormalDistribution.html.

[40] E Weisstein. “Student’s t-Distribution.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/Studentst-Distribution.html.

Appendices

Appendix A

Statistical Analysis

In Chapter 7, the discussion did not concentrate on actual analysis, but rather the method of analysis (using Encryption with 128-bit Keys as an example). The following sections contain the analysis that was performed for the remaining experiments.

A.1 Encryption with 192-bit Keys

A.1.1 Data

Language	*Mean Time*
C	0.000437662
Java	0.000440674
JavaScript	0.027895508
Common Lisp	0.087386475
Perl	0.023798828
Matlab	0.442708932

Table A.1: Data Set generated from Encryption with 192-bit Keys

A.1.2 Hypothesis

H_0: $\mu_C = \mu_{Java} = \mu_{JavaScript} = \mu_{Perl} = \mu_{Matlab} = \mu_{Common\ Lisp}$
for the Encryption with 192-bit Key test
H_1: At least two are different.

A.1.3 Analysis of Variance Tables

Source	Sum of Squares	df	Mean Square
Column Sum of Squares(CSS)	18.99377281	5	3.798754562
Residual Sum of Squares(RSS)	0.027913991	762	3.66325×10^{-5}
Total Sum of Squares (TSS)	19.0216868	767	

Table A.2: Analysis of Variance Table for Encryption with 192-bit Keys

$\tilde{F}$ = 103698.9284.

A.1.4 Least Significant Difference and Bonferroni Tests

Least Significant Difference bound:

$|\ \hat{\mu}_j - \hat{\mu}_k\ | > 0.001485190$

Bonferroni Test bound:

$|\ \hat{\mu}_j - \hat{\mu}_k\ | > 0.002222987$

Lang j	Lang k	$\|\hat{\mu}_j$ - $\hat{\mu}_k\|$	LSD	BT
C	Java	3.05176×10^{-6}	False	False
C	JavaScript	0.027457886	True	True
C	Common Lisp	0.086948853	True	True
C	Perl	0.023361206	True	True
C	Matlab	0.442271310	True	True
Java	JavaScript	0.027454834	True	True
Java	Common Lisp	0.086945801	True	True
Java	Perl	0.023358154	True	True
Java	Matlab	0.442268258	True	True
JavaScript	Common Lisp	0.059490967	True	True
JavaScript	Perl	0.004096680	True	True
JavaScript	Matlab	0.414813424	True	True
Common Lisp	Perl	0.063587647	True	True
Common Lisp	Matlab	0.355322457	True	True
Perl	Matlab	0.418910104	True	True

Table A.3: Output of Least Significance Difference and Bonferroni Tests for Encryption with 192-bit Keys

A.1.5 Test Results

As can be seen from Table A.3, the only languages that can be considered equivalent for Encryption with 192-bit keys, according the statistical analysis, are the C and Java implementations — there is insufficient evidence at the 5% level to suggest the other implementations are similar.

A.2 Encryption with 256-bit Keys

A.2.1 Data

Language	*Mean Time*
C	0.000435181
Java	0.000429077
JavaScript	0.032866821
Common Lisp	0.101591797
Perl	0.027548218
Matlab	0.627511749

Table A.4: Data Set generated from Encryption with 256-bit Keys

A.2.2 Hypothesis

H_0: $\mu_C = \mu_{Java} = \mu_{JavaScript} = \mu_{Perl} = \mu_{Matlab} = \mu_{Common\ Lisp}$
for the Encryption with 256-bit Key test
H_1: At least two are different.

A.2.3 Analysis of Variance Tables

Source	Sum of Squares	df	Mean Square
Column Sum of Squares(CSS)	38.63217763	5	7.726435527
Residual Sum of Squares(RSS)	0.053317636	762	6.99707×10^{-5}
Total Sum of Squares (TSS)	38.68549527	767	

Table A.5: Analysis of Variance Table for Encryption with 256-bit Keys

$\tilde{F}$ = 110423.9482

A.2.4 Least Significant Difference and Bonferroni Tests

Least Significant Difference bound:

$| \hat{\mu}_j - \hat{\mu}_k | > 0.002052610$

Bonferroni Test bound:

$| \hat{\mu}_j - \hat{\mu}_k | > 0.003072285$

Lang j	Lang k	$\lvert\hat{\mu}_j$ - $\hat{\mu}_k\rvert$	LSD	BT
C	Java	6.10352×10^{-6}	False	False
C	JavaScript	0.032431641	True	True
C	Common Lisp	0.101156617	True	True
C	Perl	0.027113037	True	True
C	Matlab	0.627076568	True	True
Java	JavaScript	0.032437744	True	True
Java	Common Lisp	0.101162720	True	True
Java	Perl	0.027119141	True	True
Java	Matlab	0.627082672	True	True
JavaScript	Common Lisp	0.068724976	True	True
JavaScript	Perl	0.005318604	True	True
JavaScript	Matlab	0.594644928	True	True
Common Lisp	Perl	0.074043579	True	True
Common Lisp	Matlab	0.525919952	True	True
Perl	Matlab	0.599963531	True	True

Table A.6: Output of Least Significance Difference and Bonferroni Tests for Encryption with 256-bit Keys

A.2.5 Test Results

As can be seen from Table A.6, the only languages that can be considered equivalent for Encryption with 256-bit keys, according the statistical analysis, are the C and Java implementations — there is insufficient evidence at the 5% level to suggest the other implementations are similar.

A.3 Decryption with 128-bit Keys

A.3.1 Data

Language	*Mean Time*
C	0.001325684
Java	0.000701294
JavaScript	0.092683716
Common Lisp	0.297310176
Perl	0.074577026
Matlab	2.153083282

Table A.7: Data Set generated from Decryption with 128-bit Keys

A.3.2 Hypothesis

H_0: $\mu_C = \mu_{Java} = \mu_{JavaScript} = \mu_{Perl} = \mu_{Matlab} = \mu_{Common\ Lisp}$
for the Decryption with 128-bit Key test
H_1: At least two are different.

A.3.3 Analysis of Variance Tables

Source	Sum of Squares	df	Mean Square
Column Sum of Squares(CSS)	460.0994521	5	92.01989041
Residual Sum of Squares(RSS)	0.012808374	762	1.68089×10^{-5}
Total Sum of Squares (TSS)	460.1122604	767	

Table A.8: Analysis of Variance Table for Decryption with 128-bit Keys

$\tilde{F}$ = 5474477.767

A.3.4 Least Significant Difference and Bonferroni Tests

Least Significant Difference bound:

$| \hat{\mu}_j - \hat{\mu}_k | > 0.001006046$

Bonferroni Test bound:

$| \hat{\mu}_j - \hat{\mu}_k | > 0.001505820$

Lang j	Lang k	$\|\hat{\mu}_j$ - $\hat{\mu}_k\|$	LSD	BT
C	Java	0.000624390	False	False
C	JavaScript	0.091358032	True	True
C	Common Lisp	0.295984492	True	True
C	Perl	0.073251343	True	True
C	Matlab	2.151757599	True	True
Java	JavaScript	0.091982422	True	True
Java	Common Lisp	0.296608882	True	True
Java	Perl	0.073875732	True	True
Java	Matlab	2.152381988	True	True
JavaScript	Common Lisp	0.204626460	True	True
JavaScript	Perl	0.018106689	True	True
JavaScript	Matlab	2.060399567	True	True
Common Lisp	Perl	0.222733150	True	True
Common Lisp	Matlab	1.855773106	True	True
Perl	Matlab	2.078506256	True	True

Table A.9: Output of Least Significance Difference and Bonferroni Tests for Decryption with 128-bit Keys

A.3.5 Test Results

As can be seen from Table A.9, the only languages that can be considered equivalent for Decryption with 128-bit keys, according the statistical analysis, are the C and Java implementations — there is insufficient evidence at the 5% level to suggest the other implementations are similar.

A.4 Decryption with 192-bit Keys

A.4.1 Data

Language	*Mean Time*
C	0.001559448
Java	0.000794678
JavaScript	0.112312012
Common Lisp	0.361439818
Perl	0.089953613
Matlab	2.143793864

Table A.10: Data Set generated from Decryption with 192-bit Keys

A.4.2 Hypothesis

H_0: μ_C = μ_{Java} = $\mu_{JavaScript}$ = μ_{Perl} = μ_{Matlab} = $\mu_{Common\ Lisp}$
for the Decryption with 192-bit Key test
H_1: At least two are different.

A.4.3 Analysis of Variance Tables

Source	Sum of Squares	df	Mean Square
Column Sum of Squares(CSS)	450.9843591	5	90.19687183
Residual Sum of Squares(RSS)	0.024480372	762	3.21265×10^{-5}
Total Sum of Squares (TSS)	451.0088395	767	

Table A.11: Analysis of Variance Table for Decryption with 192-bit Keys

$\tilde{F}$ = 2807555.987

A.4.4 Least Significant Difference and Bonferroni Tests

Least Significant Difference bound:

$|\ \hat{\mu}_j$ - $\hat{\mu}_k\ | > 0.001390849$

Bonferroni Test bound:

$|\ \hat{\mu}_j$ - $\hat{\mu}_k\ | > 0.002081781$

Lang j	Lang k	$\|\hat{\mu}_j$ - $\hat{\mu}_k\|$	LSD	BT
C	Java	0.000764771	False	False
C	JavaScript	0.110752563	True	True
C	Common Lisp	0.359880369	True	True
C	Perl	0.088394165	True	True
C	Matlab	2.142234416	True	True
Java	JavaScript	0.111517334	True	True
Java	Common Lisp	0.360645140	True	True
Java	Perl	0.089158936	True	True
Java	Matlab	2.142999186	True	True
JavaScript	Common Lisp	0.249127806	True	True
JavaScript	Perl	0.022358398	True	True
JavaScript	Matlab	2.031481852	True	True
Common Lisp	Perl	0.271486204	True	True
Common Lisp	Matlab	1.782354046	True	True
Perl	Matlab	2.053840251	True	True

Table A.12: Output of Least Significance Difference and Bonferroni Tests for Decryption with 192-bit Keys

A.4.5 Test Results

As can be seen from Table A.12, the only languages that can be considered equivalent for Decryption with 192-bit keys, according the statistical analysis, are the C and Java implementations — there is insufficient evidence at the 5% level to suggest the other implementations are similar.

A.5 Decryption with 256-bit Keys

A.5.1 Data

Language	*Mean Time*
C	0.001905518
Java	0.000926514
JavaScript	0.130484619
Common Lisp	0.420760496
Perl	0.104390259
Matlab	3.054164864

Table A.13: Data Set generated from Decryption with 256-bit Keys

A.5.2 Hypothesis

H$_0$: μ_C = μ_{Java} = $\mu_{JavaScript}$ = μ_{Perl} = μ_{Matlab} = $\mu_{Common\ Lisp}$
for the Decryption with 256-bit Key test
H$_1$: At least two are different.

A.5.3 Analysis of Variance Tables

Source	Sum of Squares	df	Mean Square
Column Sum of Squares(CSS)	926.1590273	5	185.2318055
Residual Sum of Squares(RSS)	0.02807659	762	3.68459×10^{-5}
Total Sum of Squares (TSS)	926.1871039	767	

Table A.14: Analysis of Variance Table for Decryption with 256-bit Keys

$\tilde{F}$ = 5027200.035

A.5.4 Least Significant Difference and Bonferroni Tests

Least Significant Difference bound:

$| \hat{\mu}_j$ - $\hat{\mu}_k | > 0.001489509$

Bonferroni Test bound:

$| \hat{\mu}_j$ - $\hat{\mu}_k | > 0.002229452$

Lang j	Lang k	$\|\hat{\mu}_j$ - $\hat{\mu}_k\|$	LSD	BT
C	Java	0.000979004	False	False
C	JavaScript	0.128579102	True	True
C	Common Lisp	0.418854978	True	True
C	Perl	0.102484741	True	True
C	Matlab	3.052259346	True	True
Java	JavaScript	0.129558105	True	True
Java	Common Lisp	0.419833982	True	True
Java	Perl	0.103463745	True	True
Java	Matlab	3.053238350	True	True
JavaScript	Common Lisp	0.290275877	True	True
JavaScript	Perl	0.026094360	True	True
JavaScript	Matlab	2.923680245	True	True
Common Lisp	Perl	0.316370237	True	True
Common Lisp	Matlab	2.633404368	True	True
Perl	Matlab	2.949774605	True	True

Table A.15: Output of Least Significance Difference and Bonferroni Tests for Decryption with 256-bit Keys

A.5.5 Test Results

As can be seen from Table A.15, the only languages that can be considered equivalent for Decryption with 256-bit keys, according the statistical analysis, are the C and Java implementations — there is insufficient evidence at the 5% level to suggest the other implementations are similar.

Appendix B

Code

The following Appendix contain the listings of the source codes of each C version, split separately into implementation and application. This listing, along with the other implementations, are also available on the attached CD, along with a working version of each implementation.

B.1 C Code

About The Source

There are two working versions of this implementation, one for Windows systems and one for Unix type systems. The Windows implementation can be found in *Impl\C\Win* and the Unix implementation in *Impl\C\Unix*.

The original source is located separately from the implementations and are located as mentioned in each subsection below. The Windows implementation was compiled using the Windows gcc compiler from MinGW [20], whilst the Unix version was compiled using the gcc compiler located on the University of Bath campus network.

B.1.1 Algorithm

Source Location: *Code\C\Algorithm*

```
/*
 * C version of Rijndael Encoding Algorithm
 * Author: Thierry Draper
 */
#include "Rijndael.h"

/* Encryption part of code */

// Encrypt the passed input
unsigned char *encrypt(unsigned char *Input, int BlockLength, unsigned char *CipherKey, int KeyLength) {
  // Variable defs
  int Nb;
  int Nk;
  int Nr;

  // Variable Set-ups
  int i = 0;
  int j;
  unsigned char *Output;
  unsigned char *SBox = (char *)calloc(256, sizeof(char));

  // Define 2D-arrays for both the state and Cipher Key
  unsigned char **State;
  unsigned char **ExpandedKey;

  // Define important variables for algorithm
  Nb = (BlockLength * 8) / 32;
  Nk = (KeyLength * 8) / 32;
  Nr = GetNumberOfRounds(Nb, Nk);

  // Define arrays, using calloc calls
  Output = (unsigned char *)calloc(4*Nb, sizeof(unsigned char));
  State = (unsigned char **)calloc(4, sizeof(unsigned char *));
```

```
State[0] = (unsigned char *)calloc(Nb, sizeof(unsigned char));
State[1] = (unsigned char *)calloc(Nb, sizeof(unsigned char));
State[2] = (unsigned char *)calloc(Nb, sizeof(unsigned char));
State[3] = (unsigned char *)calloc(Nb, sizeof(unsigned char));

ExpandedKey = (unsigned char **)calloc(4, sizeof(unsigned char *));
ExpandedKey[0] = (unsigned char *)calloc(Nk*(Nr+1), sizeof(unsigned char));
ExpandedKey[1] = (unsigned char *)calloc(Nk*(Nr+1), sizeof(unsigned char));
ExpandedKey[2] = (unsigned char *)calloc(Nk*(Nr+1), sizeof(unsigned char));
ExpandedKey[3] = (unsigned char *)calloc(Nk*(Nr+1), sizeof(unsigned char));

// Create the SBox
createSBox(SBox);

// Convert input to initial state (1D array to 2D array)
for (; i < Nb; i++) {
  State[0][i] = Input[4*i];
  State[1][i] = Input[(4*i)+1];
  State[2][i] = Input[(4*i)+2];
  State[3][i] = Input[(4*i)+3];
}

// Perform the rounds
KeyExpansion(CipherKey, ExpandedKey, Nk, Nb, Nr, SBox);
AddRoundKey(State, ExpandedKey, Nb, 0);
for (i = 0; i < (Nr-1); i++) {
  // Call the round function
  Round(State, ExpandedKey, Nb, SBox, i+1);
}
FinalRound(State, ExpandedKey, Nb, SBox, i+1);

// Re-convert final state from 2D to 1D
for (i = 0; i < Nb; i++) {
  Output[4*i] = State[0][i];
  Output[(4*i)+1] = State[1][i];
  Output[(4*i)+2] = State[2][i];
  Output[(4*i)+3] = State[3][i];
  }

  return Output;
}

// Create the SBox
void createSBox(unsigned char *SBox) {
  SBox[0]   = 0x63; SBox[1]   = 0x7c; SBox[2]   = 0x77; SBox[3]   = 0x7b;
  SBox[4]   = 0xf2; SBox[5]   = 0x6b; SBox[6]   = 0x6f; SBox[7]   = 0xc5;
  SBox[8]   = 0x30; SBox[9]   = 0x01; SBox[10]  = 0x67; SBox[11]  = 0x2b;
  SBox[12]  = 0xfe; SBox[13]  = 0xd7; SBox[14]  = 0xab; SBox[15]  = 0x76;
  SBox[16]  = 0xca; SBox[17]  = 0x82; SBox[18]  = 0xc9; SBox[19]  = 0x7d;
  SBox[20]  = 0xfa; SBox[21]  = 0x59; SBox[22]  = 0x47; SBox[23]  = 0xf0;
  SBox[24]  = 0xad; SBox[25]  = 0xd4; SBox[26]  = 0xa2; SBox[27]  = 0xaf;
  SBox[28]  = 0x9c; SBox[29]  = 0xa4; SBox[30]  = 0x72; SBox[31]  = 0xc0;
  SBox[32]  = 0xb7; SBox[33]  = 0xfd; SBox[34]  = 0x93; SBox[35]  = 0x26;
  SBox[36]  = 0x36; SBox[37]  = 0x3f; SBox[38]  = 0xf7; SBox[39]  = 0xcc;
  SBox[40]  = 0x34; SBox[41]  = 0xa5; SBox[42]  = 0xe5; SBox[43]  = 0xf1;
  SBox[44]  = 0x71; SBox[45]  = 0xd8; SBox[46]  = 0x31; SBox[47]  = 0x15;
  SBox[48]  = 0x04; SBox[49]  = 0xc7; SBox[50]  = 0x23; SBox[51]  = 0xc3;
  SBox[52]  = 0x18; SBox[53]  = 0x96; SBox[54]  = 0x05; SBox[55]  = 0x9a;
  SBox[56]  = 0x07; SBox[57]  = 0x12; SBox[58]  = 0x80; SBox[59]  = 0xe2;
  SBox[60]  = 0xeb; SBox[61]  = 0x27; SBox[62]  = 0xb2; SBox[63]  = 0x75;
  SBox[64]  = 0x09; SBox[65]  = 0x83; SBox[66]  = 0x2c; SBox[67]  = 0x1a;
  SBox[68]  = 0x1b; SBox[69]  = 0x6e; SBox[70]  = 0x5a; SBox[71]  = 0xa0;
  SBox[72]  = 0x52; SBox[73]  = 0x3b; SBox[74]  = 0xd6; SBox[75]  = 0xb3;
  SBox[76]  = 0x29; SBox[77]  = 0xe3; SBox[78]  = 0x2f; SBox[79]  = 0x84;
  SBox[80]  = 0x53; SBox[81]  = 0xd1; SBox[82]  = 0x00; SBox[83]  = 0xed;
  SBox[84]  = 0x20; SBox[85]  = 0xfc; SBox[86]  = 0xb1; SBox[87]  = 0x5b;
  SBox[88]  = 0x6a; SBox[89]  = 0xcb; SBox[90]  = 0xbe; SBox[91]  = 0x39;
  SBox[92]  = 0x4a; SBox[93]  = 0x4c; SBox[94]  = 0x58; SBox[95]  = 0xcf;
  SBox[96]  = 0xd0; SBox[97]  = 0xef; SBox[98]  = 0xaa; SBox[99]  = 0xfb;
  SBox[100] = 0x43; SBox[101] = 0x4d; SBox[102] = 0x33; SBox[103] = 0x85;
  SBox[104] = 0x45; SBox[105] = 0xf9; SBox[106] = 0x02; SBox[107] = 0x7f;
  SBox[108] = 0x50; SBox[109] = 0x3c; SBox[110] = 0x9f; SBox[111] = 0xa8;
  SBox[112] = 0x51; SBox[113] = 0xa3; SBox[114] = 0x40; SBox[115] = 0x8f;
  SBox[116] = 0x92; SBox[117] = 0x9d; SBox[118] = 0x38; SBox[119] = 0xf5;
  SBox[120] = 0xbc; SBox[121] = 0xb6; SBox[122] = 0xda; SBox[123] = 0x21;
  SBox[124] = 0x10; SBox[125] = 0xff; SBox[126] = 0xf3; SBox[127] = 0xd2;
  SBox[128] = 0xcd; SBox[129] = 0x0c; SBox[130] = 0x13; SBox[131] = 0xec;
  SBox[132] = 0x5f; SBox[133] = 0x97; SBox[134] = 0x44; SBox[135] = 0x17;
  SBox[136] = 0xc4; SBox[137] = 0xa7; SBox[138] = 0x7e; SBox[139] = 0x3d;
  SBox[140] = 0x64; SBox[141] = 0x5d; SBox[142] = 0x19; SBox[143] = 0x73;
  SBox[144] = 0x60; SBox[145] = 0x81; SBox[146] = 0x4f; SBox[147] = 0xdc;
  SBox[148] = 0x22; SBox[149] = 0x2a; SBox[150] = 0x90; SBox[151] = 0x88;
  SBox[152] = 0x46; SBox[153] = 0xee; SBox[154] = 0xb8; SBox[155] = 0x14;
  SBox[156] = 0xde; SBox[157] = 0x5e; SBox[158] = 0x0b; SBox[159] = 0xdb;
  SBox[160] = 0xe0; SBox[161] = 0x32; SBox[162] = 0x3a; SBox[163] = 0x0a;
  SBox[164] = 0x49; SBox[165] = 0x06; SBox[166] = 0x24; SBox[167] = 0x5c;
  SBox[168] = 0xc2; SBox[169] = 0xd3; SBox[170] = 0xac; SBox[171] = 0x62;
  SBox[172] = 0x91; SBox[173] = 0x95; SBox[174] = 0xe4; SBox[175] = 0x79;
  SBox[176] = 0xe7; SBox[177] = 0xc8; SBox[178] = 0x37; SBox[179] = 0x6d;
  SBox[180] = 0x8d; SBox[181] = 0xd5; SBox[182] = 0x4e; SBox[183] = 0xa9;
  SBox[184] = 0x6c; SBox[185] = 0x56; SBox[186] = 0xf4; SBox[187] = 0xea;
  SBox[188] = 0x65; SBox[189] = 0x7a; SBox[190] = 0xae; SBox[191] = 0x08;
  SBox[192] = 0xba; SBox[193] = 0x78; SBox[194] = 0x25; SBox[195] = 0x2e;
  SBox[196] = 0x1c; SBox[197] = 0xa6; SBox[198] = 0xb4; SBox[199] = 0xc6;
  SBox[200] = 0xe8; SBox[201] = 0xdd; SBox[202] = 0x74; SBox[203] = 0x1f;
  SBox[204] = 0x4b; SBox[205] = 0xbd; SBox[206] = 0x8b; SBox[207] = 0x8a;
  SBox[208] = 0x70; SBox[209] = 0x3e; SBox[210] = 0xb5; SBox[211] = 0x66;
  SBox[212] = 0x48; SBox[213] = 0x03; SBox[214] = 0xf6; SBox[215] = 0x0e;
  SBox[216] = 0x61; SBox[217] = 0x35; SBox[218] = 0x57; SBox[219] = 0xb9;
  SBox[220] = 0x86; SBox[221] = 0xc1; SBox[222] = 0x1d; SBox[223] = 0x9e;
  SBox[224] = 0xe1; SBox[225] = 0xf8; SBox[226] = 0x98; SBox[227] = 0x11;
  SBox[228] = 0x69; SBox[229] = 0xd9; SBox[230] = 0x8e; SBox[231] = 0x94;
  SBox[232] = 0x9b; SBox[233] = 0x1e; SBox[234] = 0x87; SBox[235] = 0xe9;
  SBox[236] = 0xce; SBox[237] = 0x55; SBox[238] = 0x28; SBox[239] = 0xdf;
  SBox[240] = 0x8c; SBox[241] = 0xa1; SBox[242] = 0x89; SBox[243] = 0x0d;
  SBox[244] = 0xbf; SBox[245] = 0xe6; SBox[246] = 0x42; SBox[247] = 0x68;
  SBox[248] = 0x41; SBox[249] = 0x99; SBox[250] = 0x2d; SBox[251] = 0x0f;
  SBox[252] = 0xb0; SBox[253] = 0x54; SBox[254] = 0xbb; SBox[255] = 0x16;
}

// Perform a Round
```

```
void Round(unsigned char **State, unsigned char **ExpandedKey, int Nb, unsigned char *SBox, int round) {
  // A Round consists of the following function calls
  SubBytes(State, Nb, SBox);
  ShiftRows(State, Nb);
  MixColumns(State, Nb);
  AddRoundKey(State, ExpandedKey, Nb, round);
}

// Final Round of process - almost the same as the rest of the rounds, but without the call to MixColumn
void FinalRound(unsigned char **State, unsigned char **ExpandedKey, int Nb, unsigned char *SBox, int round) {
  SubBytes(State, Nb, SBox);
  ShiftRows(State, Nb);
  AddRoundKey(State, ExpandedKey, Nb, round);
}

// Produce the expanded key from the cipher key
void KeyExpansion(unsigned char *CipherKey, unsigned char **ExpandedKey, int Nk, int Nb, int Nr, unsigned char *SBox) {
  int i = 0;
  unsigned char *temp = (char *)calloc(4, sizeof(char));
  unsigned char rc = 1;
  unsigned char rcXor;

  // Map the Cipher Key to the first Nk 32-bit words of the Expanded Key
  for (; i < Nk; i++) {
    ExpandedKey[0][i] = CipherKey[4*i];
    ExpandedKey[1][i] = CipherKey[(4*i)+1];
    ExpandedKey[2][i] = CipherKey[(4*i)+2];
    ExpandedKey[3][i] = CipherKey[(4*i)+3];
  }

  // Now recursively define the Expanded Key in terms of 32-bit words of smaller indicies
  for (; i < (Nb * (Nr + 1)); i++) {
    temp[0] = ExpandedKey[0][i-1];
    temp[1] = ExpandedKey[1][i-1];
    temp[2] = ExpandedKey[2][i-1];
    temp[3] = ExpandedKey[3][i-1];
    if (i % Nk == 0) {
      // Mutate the above word
      RotByte(temp);
      SubByte(temp, SBox);
      temp[0] ^= rc;

      rcXor = (rc & 0x80);
      rc <<= 1;
      if (rcXor)
        rc ^= 0x1b;

    } else if (Nk > 6 && i % Nk == 4) {
      // Special operation for when key size > 192
      SubByte(temp, SBox);
    }

    ExpandedKey[0][i] = ExpandedKey[0][i - Nk] ^ temp[0];
    ExpandedKey[1][i] = ExpandedKey[1][i - Nk] ^ temp[1];
    ExpandedKey[2][i] = ExpandedKey[2][i - Nk] ^ temp[2];
    ExpandedKey[3][i] = ExpandedKey[3][i - Nk] ^ temp[3];
  }
}

// Non-linear Byte Substitution
void SubBytes(unsigned char **State, int Nb, unsigned char *SBox) {
  int i = 0, j;

  // Perform the following on each byte of the state
  for (; i < 4; i++)
    for (j = 0; j < Nb; j++)
      State[i][j] = SBox[State[i][j]];
}

// Shift rows of the state round cyclicaclly
void ShiftRows(unsigned char **State, int Nb) {
  int i = 1, j;
  int shift;
  unsigned char *temp_bytes;

  // Loop through rows 1-3, cyclically shifting
  for (; i < 4; i++) {
    // Get chars from end of string (which will move to the front of the string in shifted version)
    shift = ShiftOffset(Nb, i);
    temp_bytes = (unsigned char *)calloc(Nb, sizeof(unsigned char));

    // Get initial characters which will be over-written by shift
    for (j = 0; j < shift; j++)
      temp_bytes[Nb-shift+j] = State[i][j];

    // Shift characters right
    for (j = shift; j < Nb; j++)
      temp_bytes[j-shift] = State[i][j];

    State[i][0] = temp_bytes[0];
    State[i][1] = temp_bytes[1];
    State[i][2] = temp_bytes[2];
    State[i][3] = temp_bytes[3];
  }
}

// Play with the State a little more
void MixColumns(unsigned char **State, int Nb) {
  int i = 0;
  unsigned char *s = (unsigned char *)calloc(4, sizeof(unsigned char));

  for (; i < Nb; i++) {
    // Get current values, before State gets mutated
    s[0] = State[0][i];
    s[1] = State[1][i];
    s[2] = State[2][i];
    s[3] = State[3][i];

    // Perform matrix multiplication
    State[0][i] = gfMult(0x02, s[0], 0x11b) ^ gfMult(0x03, s[1], 0x11b) ^ s[2] ^ s[3];
    State[1][i] = s[0] ^ gfMult(0x02, s[1], 0x11b) ^ gfMult(0x03, s[2], 0x11b) ^ s[3];
    State[2][i] = s[0] ^ s[1] ^ gfMult(0x02, s[2], 0x11b) ^ gfMult(0x03, s[3], 0x11b);
    State[3][i] = gfMult(0x03, s[0], 0x11b) ^ s[1] ^ s[2] ^ gfMult(0x02, s[3], 0x11b);
  }
}

// Apply the key to the State
void AddRoundKey(unsigned char **State, unsigned char **ExpandedKey, int Nb, int round) {
  int i = 0, j;

  for (i = 0; i < 4; i++)
    for (j = 0; j < Nb; j++)
      State[i][j] ^= ExpandedKey[i][(Nb*round)+j];
}

/* End of encrypt section of file */


/* Decryption part of code*/

// Decrypt the passed input
unsigned char *decrypt(unsigned char *Input, int BlockLength, unsigned char *CipherKey, int KeyLength) {
  // Variable defs
  int Nb;
  int Nk;
  int Nr;

  // Variable Set-ups
  int i = 0;
  unsigned char *Output;
  unsigned char *InvSBox = (char *)calloc(256, sizeof(char));
  unsigned char *SBox = (char *)calloc(256, sizeof(char));

  // Define 2D-arrays for both the state and Cipher Key
  unsigned char **State;
  unsigned char **InvExpandedKey;
```

```
// Define important variables for algorithm
Nb = (BlockLength * 8) / 32;
Nk = (KeyLength * 8) / 32;
Nr = GetNumberOfRounds(Nb, Nk);

// Define arrays, using calloc calls
Output = (unsigned char *)calloc(4*Nb, sizeof(unsigned char));
State = (unsigned char **)calloc(4, sizeof(unsigned char *));
State[0] = (unsigned char *)calloc(Nb, sizeof(unsigned char));
State[1] = (unsigned char *)calloc(Nb, sizeof(unsigned char));
State[2] = (unsigned char *)calloc(Nb, sizeof(unsigned char));
State[3] = (unsigned char *)calloc(Nb, sizeof(unsigned char));

InvExpandedKey = (unsigned char **)calloc(4, sizeof(unsigned char *));
InvExpandedKey[0] = (unsigned char *)calloc(Nb*(Nr+1), sizeof(unsigned char));
InvExpandedKey[1] = (unsigned char *)calloc(Nk*(Nr+1), sizeof(unsigned char));
InvExpandedKey[2] = (unsigned char *)calloc(Nk*(Nr+1), sizeof(unsigned char));
InvExpandedKey[3] = (unsigned char *)calloc(Nk*(Nr+1), sizeof(unsigned char));

// Create the SBox
createInvSBox(InvSBox);
createSBox(SBox);

// Convert input to initial state (1D array to 2D array)
for (; i < Nb; i++) {
  State[0][i] = Input[4*i];
  State[1][i] = Input[(4*i)+1];
  State[2][i] = Input[(4*i)+2];
  State[3][i] = Input[(4*i)+3];
}

// Perform the rounds
InvKeyExpansion(CipherKey, InvExpandedKey, Nk, Nb, Nr, SBox);
InvAddRoundKey(State, InvExpandedKey, Nb*Nr, Nb);
for (i = Nr - 1; i > 0; i--) {
  // Call the round function
  InvRound(State, InvExpandedKey, Nb*i, Nb, InvSBox);
  }
  InvFinalRound(State, InvExpandedKey, Nb, InvSBox);

  // Re-convert final state from 2D to 1D
  for (i = 0; i < Nb; i++) {
    Output[4*i] = State[0][i];
    Output[(4*i)+1] = State[1][i];
    Output[(4*i)+2] = State[2][i];
    Output[(4*i)+3] = State[3][i];
  }

  return Output;
}

// Create the Inverse SBox
void createInvSBox(unsigned char *InvSBox) {
  InvSBox[0]   = 0x52; InvSBox[1]   = 0x09; InvSBox[2]   = 0x6a; InvSBox[3]   = 0xd5;
  InvSBox[4]   = 0x30; InvSBox[5]   = 0x36; InvSBox[6]   = 0xa5; InvSBox[7]   = 0x38;
  InvSBox[8]   = 0xbf; InvSBox[9]   = 0x40; InvSBox[10]  = 0xa3; InvSBox[11]  = 0x9e;
  InvSBox[12]  = 0x81; InvSBox[13]  = 0xf3; InvSBox[14]  = 0xd7; InvSBox[15]  = 0xfb;
  InvSBox[16]  = 0x7c; InvSBox[17]  = 0xe3; InvSBox[18]  = 0x39; InvSBox[19]  = 0x82;
  InvSBox[20]  = 0x9b; InvSBox[21]  = 0x2f; InvSBox[22]  = 0xff; InvSBox[23]  = 0x87;
  InvSBox[24]  = 0x34; InvSBox[25]  = 0x8e; InvSBox[26]  = 0x43; InvSBox[27]  = 0x44;
  InvSBox[28]  = 0xc4; InvSBox[29]  = 0xde; InvSBox[30]  = 0xe9; InvSBox[31]  = 0xcb;
  InvSBox[32]  = 0x54; InvSBox[33]  = 0x7b; InvSBox[34]  = 0x94; InvSBox[35]  = 0x32;
  InvSBox[36]  = 0xa6; InvSBox[37]  = 0xc2; InvSBox[38]  = 0x23; InvSBox[39]  = 0x3d;
  InvSBox[40]  = 0xee; InvSBox[41]  = 0x4c; InvSBox[42]  = 0x95; InvSBox[43]  = 0x0b;
  InvSBox[44]  = 0x42; InvSBox[45]  = 0xfa; InvSBox[46]  = 0xc3; InvSBox[47]  = 0x4e;
  InvSBox[48]  = 0x08; InvSBox[49]  = 0x2e; InvSBox[50]  = 0xa1; InvSBox[51]  = 0x66;
  InvSBox[52]  = 0x28; InvSBox[53]  = 0xd9; InvSBox[54]  = 0x24; InvSBox[55]  = 0xb2;
  InvSBox[56]  = 0x76; InvSBox[57]  = 0x5b; InvSBox[58]  = 0xa2; InvSBox[59]  = 0x49;
  InvSBox[60]  = 0x6d; InvSBox[61]  = 0x8b; InvSBox[62]  = 0xd1; InvSBox[63]  = 0x25;
  InvSBox[64]  = 0x72; InvSBox[65]  = 0xf8; InvSBox[66]  = 0xf6; InvSBox[67]  = 0x64;
  InvSBox[68]  = 0x86; InvSBox[69]  = 0x68; InvSBox[70]  = 0x98; InvSBox[71]  = 0x16;
  InvSBox[72]  = 0xd4; InvSBox[73]  = 0xa4; InvSBox[74]  = 0x5c; InvSBox[75]  = 0xcc;
  InvSBox[76]  = 0x5d; InvSBox[77]  = 0x65; InvSBox[78]  = 0xb6; InvSBox[79]  = 0x92;
  InvSBox[80]  = 0x6c; InvSBox[81]  = 0x70; InvSBox[82]  = 0x48; InvSBox[83]  = 0x50;
```

```
InvSBox[84]  = 0xfd; InvSBox[85]  = 0xed; InvSBox[86]  = 0xb9; InvSBox[87]  = 0xda;
InvSBox[88]  = 0x5e; InvSBox[89]  = 0x15; InvSBox[90]  = 0x46; InvSBox[91]  = 0x57;
InvSBox[92]  = 0xa7; InvSBox[93]  = 0x8d; InvSBox[94]  = 0x9d; InvSBox[95]  = 0x84;
InvSBox[96]  = 0x90; InvSBox[97]  = 0xd8; InvSBox[98]  = 0xab; InvSBox[99]  = 0x00;
InvSBox[100] = 0x8c; InvSBox[101] = 0xbc; InvSBox[102] = 0xd3; InvSBox[103] = 0x0a;
InvSBox[104] = 0xf7; InvSBox[105] = 0xe4; InvSBox[106] = 0x58; InvSBox[107] = 0x05;
InvSBox[108] = 0xb8; InvSBox[109] = 0xb3; InvSBox[110] = 0x45; InvSBox[111] = 0x06;
InvSBox[112] = 0xd0; InvSBox[113] = 0x2c; InvSBox[114] = 0x1e; InvSBox[115] = 0x8f;
InvSBox[116] = 0xca; InvSBox[117] = 0x3f; InvSBox[118] = 0x0f; InvSBox[119] = 0x02;
InvSBox[120] = 0xc1; InvSBox[121] = 0xaf; InvSBox[122] = 0xbd; InvSBox[123] = 0x03;
InvSBox[124] = 0x01; InvSBox[125] = 0x13; InvSBox[126] = 0x8a; InvSBox[127] = 0x6b;
InvSBox[128] = 0x3a; InvSBox[129] = 0x91; InvSBox[130] = 0x11; InvSBox[131] = 0x41;
InvSBox[132] = 0x4f; InvSBox[133] = 0x67; InvSBox[134] = 0xdc; InvSBox[135] = 0xea;
InvSBox[136] = 0x97; InvSBox[137] = 0xf2; InvSBox[138] = 0xcf; InvSBox[139] = 0xce;
InvSBox[140] = 0xf0; InvSBox[141] = 0xb4; InvSBox[142] = 0xe6; InvSBox[143] = 0x73;
InvSBox[144] = 0x96; InvSBox[145] = 0xac; InvSBox[146] = 0x74; InvSBox[147] = 0x22;
InvSBox[148] = 0xe7; InvSBox[149] = 0xad; InvSBox[150] = 0x35; InvSBox[151] = 0x85;
InvSBox[152] = 0xe2; InvSBox[153] = 0xf9; InvSBox[154] = 0x37; InvSBox[155] = 0xe8;
InvSBox[156] = 0x1c; InvSBox[157] = 0x75; InvSBox[158] = 0xdf; InvSBox[159] = 0x6e;
InvSBox[160] = 0x47; InvSBox[161] = 0xf1; InvSBox[162] = 0x1a; InvSBox[163] = 0x71;
InvSBox[164] = 0x1d; InvSBox[165] = 0x29; InvSBox[166] = 0xc5; InvSBox[167] = 0x89;
InvSBox[168] = 0x6f; InvSBox[169] = 0xb7; InvSBox[170] = 0x62; InvSBox[171] = 0x0e;
InvSBox[172] = 0xaa; InvSBox[173] = 0x18; InvSBox[174] = 0xbe; InvSBox[175] = 0x1b;
InvSBox[176] = 0xfc; InvSBox[177] = 0x56; InvSBox[178] = 0x3e; InvSBox[179] = 0x4b;
InvSBox[180] = 0xc6; InvSBox[181] = 0xd2; InvSBox[182] = 0x79; InvSBox[183] = 0x20;
InvSBox[184] = 0x9a; InvSBox[185] = 0xdb; InvSBox[186] = 0xc0; InvSBox[187] = 0xfe;
InvSBox[188] = 0x78; InvSBox[189] = 0xcd; InvSBox[190] = 0x5a; InvSBox[191] = 0xf4;
InvSBox[192] = 0x1f; InvSBox[193] = 0xdd; InvSBox[194] = 0xa8; InvSBox[195] = 0x33;
InvSBox[196] = 0x88; InvSBox[197] = 0x07; InvSBox[198] = 0xc7; InvSBox[199] = 0x31;
InvSBox[200] = 0xb1; InvSBox[201] = 0x12; InvSBox[202] = 0x10; InvSBox[203] = 0x59;
InvSBox[204] = 0x27; InvSBox[205] = 0x80; InvSBox[206] = 0xec; InvSBox[207] = 0x5f;
InvSBox[208] = 0x60; InvSBox[209] = 0x51; InvSBox[210] = 0x7f; InvSBox[211] = 0xa9;
InvSBox[212] = 0x19; InvSBox[213] = 0xb5; InvSBox[214] = 0x4a; InvSBox[215] = 0x0d;
InvSBox[216] = 0x2d; InvSBox[217] = 0xe5; InvSBox[218] = 0x7a; InvSBox[219] = 0x9f;
InvSBox[220] = 0x93; InvSBox[221] = 0xc9; InvSBox[222] = 0x9c; InvSBox[223] = 0xef;
InvSBox[224] = 0xa0; InvSBox[225] = 0xe0; InvSBox[226] = 0x3b; InvSBox[227] = 0x4d;
InvSBox[228] = 0xae; InvSBox[229] = 0x2a; InvSBox[230] = 0xf5; InvSBox[231] = 0xb0;
```

```
  InvSBox[232] = 0xc8; InvSBox[233] = 0xeb; InvSBox[234] = 0xbb; InvSBox[235] = 0x3c;
  InvSBox[236] = 0x83; InvSBox[237] = 0x53; InvSBox[238] = 0x99; InvSBox[239] = 0x61;
  InvSBox[240] = 0x17; InvSBox[241] = 0x2b; InvSBox[242] = 0x04; InvSBox[243] = 0x7e;
  InvSBox[244] = 0xba; InvSBox[245] = 0x77; InvSBox[246] = 0xd6; InvSBox[247] = 0x26;
  InvSBox[248] = 0xe1; InvSBox[249] = 0x69; InvSBox[250] = 0x14; InvSBox[251] = 0x63;
  InvSBox[252] = 0x55; InvSBox[253] = 0x21; InvSBox[254] = 0x0c; InvSBox[255] = 0x7d;
}

// Produce the expanded keyset for the decipher stage
void InvKeyExpansion(unsigned char *CipherKey, unsigned char **InvExpandedKey, int Nk, int Nb, int Nr, unsigned char *SBox) {
  int i = 1;
  KeyExpansion(CipherKey, InvExpandedKey, Nk, Nb, Nr, SBox);
  for (; i < Nr; i++)
    InvMixColumns(InvExpandedKey, Nb*i, Nb);
}

// Perform a Round
void InvRound(unsigned char **State, unsigned char **InvExpandedKey, int iekStart, int Nb, unsigned char *InvSBox) {
  // This function is the same as for encryption, just use a different SBox
  SubBytes(State, Nb, InvSBox);
  InvShiftRows(State, Nb);
  InvMixColumns(State, 0, Nb);
  InvAddRoundKey(State, InvExpandedKey, iekStart, Nb);
}

// Final Round of process - almost the same as the rest of the rounds, but without the call to MixColumn
void InvFinalRound(unsigned char **State, unsigned char **InvExpandedKey, int Nb, unsigned char *InvSBox) {
  // As noted above, same function, but different SBox
  SubBytes(State, Nb, InvSBox);
  InvShiftRows(State, Nb);
  InvAddRoundKey(State, InvExpandedKey, 0, Nb);
}

// Shift rows of the state round cyclicaclly
void InvShiftRows(unsigned char **State, int Nb) {
  int i = 1, j;
  int shift;
  unsigned char *temp_bytes;

  // Loop through rows 1-3, cyclically shifting
  for (; i < 4; i++) {
    // Get chars from end of string (which will move to the front of the string in shifted version)
    shift = ShiftOffset(Nb, i);
    temp_bytes = (unsigned char *)calloc(Nb, sizeof(unsigned char));

    // Get initial characters which will be over-written by shift
    for (j = Nb-shift; j < Nb; j++)
      temp_bytes[j-Nb+shift] = State[i][j];

    // Shift characters right
    for (j = 0; j < Nb-shift; j++)
      temp_bytes[j+shift] = State[i][j];

    State[i][0] = temp_bytes[0];
    State[i][1] = temp_bytes[1];
    State[i][2] = temp_bytes[2];
    State[i][3] = temp_bytes[3];
  }
}

// Play with the State a little more
void InvMixColumns(unsigned char **Array, int start, int Nb) {
  int i = start;
  unsigned char *a = (unsigned char *)calloc(4, sizeof(unsigned char));

  for (; i < start+Nb; i++) {
    // Get current values, before State gets mutated
    a[0] = Array[0][i];
    a[1] = Array[1][i];
    a[2] = Array[2][i];
    a[3] = Array[3][i];

    // Perform matrix multiplication
    Array[0][i] = gfMult(0x0e, a[0], 0x11b) ^ gfMult(0x0b, a[1], 0x11b) ^ gfMult(0x0d, a[2], 0x11b) ^ gfMult(0x09, a[3], 0x11b);
    Array[1][i] = gfMult(0x09, a[0], 0x11b) ˆ gfMult(0x0e, a[1], 0x11b) ˆ gfMult(0x0b, a[2], 0x11b) ˆ gfMult(0x0d, a[3], 0x11b);
    Array[2][i] = gfMult(0x0d, a[0], 0x11b) ˆ gfMult(0x09, a[1], 0x11b) ˆ gfMult(0x0e, a[2], 0x11b) ˆ gfMult(0x0b, a[3], 0x11b);
    Array[3][i] = gfMult(0x0b, a[0], 0x11b) ˆ gfMult(0x0d, a[1], 0x11b) ˆ gfMult(0x09, a[2], 0x11b) ˆ gfMult(0x0e, a[3], 0x11b);
  }
}

// Apply the key to the State
void InvAddRoundKey(unsigned char **State, unsigned char **InvExpandedKey, int iekStart, int Nb) {
  int i = 0, j;

  for (i = 0; i < 4; i++)
    for (j = iekStart; j < iekStart+Nb; j++)
      State[i][j-iekStart] ˆ= InvExpandedKey[i][j];
}

/* End of decryptinn section of file */


/* General functions used in the code */

// Given a block length and a key length, give the number of rounds to perform
int GetNumberOfRounds(int Nb, int Nk) {
  if (Nb == 8 || Nk == 8)
    return 14;
  else if (Nb == 6 || Nk == 6)
    return 12;
  else
    return 10;
}

// Return the shift offsets, for the different block lengths
int ShiftOffset(int Nb, int C) {
  int offset = 0;

  // Using the chart, find out the offset
  switch (C) {
    case 1:
      return 1;
      break;
    case 2:
      offset = 2;
      break;
  case 3:
      offset = 3;
      break;
  }

  if (Nb == 8)
    offset++;

  return offset;
}

// Perform multiplication on a finite field GF(2ˆ8)
unsigned char gfMult(unsigned char e1, unsigned char e2, unsigned short mod) {
  unsigned short result = 0x0;
  unsigned short e1Arg = (short) e1;

  gfAdd(e2 << 7, 0x80, &result, e1Arg);

  if (result > 0xff)
    gfAdd(mod << 7, 0x8000, &result, result);

  return (char) result;
}

void gfAdd(unsigned short xorValue, unsigned short bitCheck, unsigned short *result, unsigned short with) {
  int i = 0;

  for (; i < 8; i++) {
    if (bitCheck & with)
      (*result) ˆ= xorValue;

    bitCheck >>= 1;
    xorValue >>= 1;
  }
}

// Substitute each input byte with a corresponding SBox value
void SubByte(unsigned char *input, unsigned char *SBox) {
  input[0] = SBox[input[0]];
  input[1] = SBox[input[1]];
  input[2] = SBox[input[2]];
  input[3] = SBox[input[3]];
}

// Single cyclic left shift
void RotByte(unsigned char *input) {
  unsigned char temp = input[0];
  int i = 0;
  for (; i < 3; i++)
    input[i] = input[i+1];
  input[3] = temp;
}

/* End of general functions used in code */
```

B.1.2 Application

Source Location: *Code\C\App*

Encrypt

```c
/*
 * Application to use encode.c
 * Author: Thierry Draper
 */

// Include necessary functions
#include "Rijndael.h"

// Function prototypes
int file_exists(char *);
void fEncrypt(char *, char *);
int random(int);

int main(int argc, char **argv) {
  int i = 1;
  char *keyFile = NULL;

  // Loop through arguments, encrypting file
  for (; i < argc; i++) {
    if (!strcmp("-k", argv[i])) {
      // Key file passed
      keyFile = argv[++i];
    } else if (file_exists(argv[i])) {
      // If the file exists, then encrypt
      fEncrypt(argv[i], keyFile);
      printf("File ‘%s’ has successfully been encrypted to file ‘%s.enc’.\n", argv[i], argv[i]);
    } else {
      // Otherwise, display error message
      printf("Unable to encrypt file ‘%s’ - File not found.\n", argv[i]);
    }
  }
}

// Check to see if the passed file exists by trying to open it
int file_exists(char *file) {
  int returnValue = 1;
/*  FILE *fp = NULL;
  fp = freopen(file, "r", fp);
  if (fp != NULL) {
    returnValue = 1;
  }

  fclose(fp);
*/  return returnValue;
}

// Open the input file and encode in 16 byte chunks
void fEncrypt(char *file, char *keyInFile) {
  int i = 0;
  int keyLength = 0;
  int fromFile;
  unsigned char keyIn;
  unsigned char *in = (unsigned char *)calloc(16, sizeof(unsigned char));
  unsigned char *out;
  unsigned char *key = (unsigned char *)calloc(1, sizeof(unsigned char));
  char *fileOut = (char *)calloc(strlen(file)+5, sizeof(char));
  char *keyOutFile = (char *)calloc(strlen(file)+9, sizeof(char));
  FILE *fKey;
  FILE *fKeyIn;
  FILE *fIn;
  FILE *fOut;

  // Organise key management and store key for decryption
  strcpy(keyOutFile, file);
  strcat(keyOutFile, ".enc.key");
  fKey = fopen(keyOutFile, "wb");
```

```
if (keyInFile == NULL) {
  // Generate random key
  keyLength = 16;
  key = (unsigned char *)calloc(keyLength, sizeof(unsigned char));
  srand((unsigned) (time(0) * getpid()));

  for (i = 0; i < keyLength; i++) {
    key[i] = random(255);
    fwrite(key+i, sizeof(unsigned char), 1, fKey);
  }

} else {
  // Load key from file
  fKeyIn = fopen(keyInFile, "rb");
  while(!feof(fKeyIn)) {
    fromFile = fread(&keyIn, sizeof(unsigned char), 1, fKeyIn);
    if (fromFile) {
      keyLength++;
      key = (unsigned char *)realloc(key, sizeof(unsigned char)*keyLength);
      key[keyLength-1] = keyIn;
      fwrite(key+keyLength-1, sizeof(unsigned char), 1, fKey);
    }
  }
  fclose(fKeyIn);
}
fclose(fKey);

strcpy(fileOut, file);
strcat(fileOut, ".enc");
fOut = fopen(fileOut, "wb");
fIn = fopen(file, "rb");

while(!feof(fIn)) {
  fromFile = fread(in, sizeof(unsigned char), 16, fIn);
  if (fromFile) {
    out = encrypt(in, 16, key, keyLength);
    fwrite(out, sizeof(unsigned char), 16, fOut);
    }
  }

  fclose(fIn);
  fclose(fOut);
}

int random(int max) {
  float rnd = (float) rand() / (RAND_MAX + 1.0);
  return (int) max*rnd;
}
```

Decrypt

```c
/*
 * Application to use encode.c
 * Author: Thierry Draper
 */

// Include necessary functions
#include "Rijndael.h"

// Function prototypes
int file_exists(char *);
void fDecrypt(char *);
int random(int);

int main(int argc, char **argv) {
  int i = 1;

  // Loop through arguments, encrypting file
  for (; i < argc; i++) {
    if (file_exists(argv[i])) {
      // If the file exists, then encrypt
      fDecrypt(argv[i]);
      printf("File ‘%s’ has successfully been decrypted to file ‘%s.dec’.\n", argv[i], argv[i]);
    } else {
      // Otherwise, display error message
      printf("Unable to encrypt file ‘%s’ - File not found.\n", argv[i]);
    }
  }
}

// Check to see if the passed file exists by trying to open it
int file_exists(char *file) {
  int returnValue = 1;
/*  FILE *fp = NULL;
  fp = freopen(file, "r", fp);
  if (fp != NULL) {
    returnValue = 1;
  }

  fclose(fp);
*/  return returnValue;
}

// Open the input file and encode in 16 byte chunks
void fDecrypt(char *file) {
  int i = 0;
  int keyLength = 16;
  int fromFile;
  unsigned char *in = (unsigned char *)calloc(16, sizeof(unsigned char));
  unsigned char *out;
  unsigned char *key = (unsigned char *)calloc(keyLength, sizeof(unsigned char));
  char *fileOut = (char *)calloc(strlen(file)+4, sizeof(char));
  char *keyFile = (char *)calloc(strlen(file)+4, sizeof(char));
  FILE *fKey;
  FILE *fIn = fopen(file, "rb");
  FILE *fOut;

  // Get key
  strcpy(keyFile, file);
  strcat(keyFile, ".key");
  fKey = fopen(keyFile, "rb");
  for (i = 0; i < keyLength; i++)
    fread(key+i, sizeof(unsigned char), 1, fKey);
  fclose(fKey);

  strcpy(fileOut, file);
  strcat(fileOut, ".dec");
  fOut = fopen(fileOut, "wb");

  while(!feof(fIn)) {
    fromFile = fread(in, sizeof(unsigned char), 16, fIn);
    if (fromFile) {
      out = decrypt(in, 16, key, keyLength);
      fwrite(out, sizeof(unsigned char), 16, fOut);
    }
  }

  fclose(fIn);
  fclose(fOut);
}
```