Lars R. Knudsen • Matthew J.B. Robshaw

The Block Cipher Companion

Chapter 1
Introduction

As our personal and commercial transactions are routinely transmitted around the world, the need to protect information has never been greater. One of the best single line descriptions of cryptography is due to Rivest [636]:

> Cryptography is about communication in the presence of an adversary.

Over the past decades the field of cryptography has grown and matured to a remarkable degree. The title of the book you are now reading is sufficient clue to the fact that we are restricting our attention to one very small part of this research, and we will restrict our attention to one particular class of cryptographic algorithm: the *block cipher*. However, we shouldn't feel too discouraged that we will only be covering a small fraction of the cryptographic research. Block ciphers are fundamental to much of today's deployed cryptography and they are, by far, the most widely used cryptographic primitive. Indeed, even though block ciphers are useful in their own right they are also used to build other cryptographic mechanisms. In short, they are particularly versatile objects.

The *block cipher* is the narrow focus of this book and we feel that an overview of the state of the art of block cipher analysis, design, and deployment is timely. The chapters that follow are of one of two types. Several chapters provide descriptions of prominent block ciphers and give an insight into their design. Other chapters consider the role of the cryptanalyst (the "adversary") and provide an overview of some of the most important cryptanalytic methods.

The field of block cipher design was somewhat quiet after the adoption of the *Advanced Encryption Standard (AES)*, a cipher which will be described in Chap. 3. In fact some cryptographic commentators went so far as to suggest that "cryptography was dead", by which they implied that the task of algorithm design was pretty much done and dusted. However, the field of block cipher design and analysis is as strong as it ever was and we believe such work to be vital. Not only do we need to continually validate the security of the techniques we use, but new applications introduce new demands for which existing algorithms might be inadequate. In fact even recent advances in hash function cryptanalysis can be viewed as the application and extension of techniques in block cipher cryptanalysis.

L.R. Knudsen and M.J.B. Robshaw, *The Block Cipher Companion*, Information Security and Cryptography, DOI 10.1007/978-3-642-17342-4_1, © Springer-Verlag Berlin Heidelberg 2011

Of course we're not looking at block ciphers just out of curiosity; we want to do something with them. In short we want to solve some security problem, to achieve a security goal. Most books on security and more than a few on cryptography provide a list of the likely security goals for an application. The contents of such lists vary among commentators, but the goals of *confidentiality*, *integrity*, and *availability* are often highlighted. Application designers will attempt to deliver these goals by appealing to a range of *security services* and these can range widely from the technical to the administrative. Cryptography typically forms only a very small component of a security solution, but it has an important role and it is likely to provide some of the following:

- *Confidentiality*: ensuring that an adversary listening to our communication channel cannot gain information about the content of our communications.
- *Data integrity*: ensuring that any adversary with access to our communication channel is unable to manipulate the contents of some communication by unauthorized means.
- *Data origin authentication*: ensuring that any adversary with access to our communication channel is unable to modify and/or misrepresent the true origin of some communication. Some commentators observe that the property of *data origin authentication* directly implies *data integrity* since a message that has been modified has a new source.

Of course there are other services that we haven't explicitly mentioned. For instance, *entity authentication* allows for the corroboration of the identity of an entity such as a person, a computer terminal, or a credit card. We might also consider issues such as *authorization*, *confirmation*, *delegation*, and *non-repudiation* along with many others. However, there are so many good treatments of these issues in the broader cryptographic and security literature (*e.g.*, see [493]) that we will not attempt to duplicate those efforts here.

Instead our interest lies in the *cryptographic algorithm* and in how a limited number of bits and bytes might be processed under the action of a secret key. For the most part we won't be interested in how the algorithm is used or in how data or secret-key material is delivered to the algorithm. These are important issues, but they are not our concern in this book.

The popular press seems to relish the opportunity to report on exciting new "security glitches" or "breaks", particularly those involving the Internet. However, it is rare that such security weaknesses are due to the cryptographic algorithm. Of course bad algorithms will give a bad application. However, the overall success of a deployment will depend on all aspects of the engineering effort; the choice of cryptographic algorithm is often a small—but important—part of the solution. When choosing a cryptographic algorithm the two most important concerns will almost certainly be *(i)* security and *(ii)* performance. And while other business-related issues such as licensing might play a part, today we are likely to find standardised algorithms offering the best choice.

It is a testament to the skills of early algorithm designers that we have had trusted algorithms available to us for many years. Nevertheless, anyone reading the cryp-

tographic literature will be struck by the large number of algorithms that have been proposed. What is it that makes a good cryptographic algorithm? What makes one better than another?

1.1 Cryptographic Algorithms

It is customary to classify cryptographic algorithms according to how key material is used. To begin, we will isolate the class of *keyless* algorithms, where we put the cryptographic algorithms that do not use *any* key material. The classic example is the *hash function* or *message-digest* algorithm.

Assuming that key material is used, and further assuming that the communication or interaction we wish to protect has multiple participants, we consider two additional classes of cryptographic algorithms. For some algorithms all participants share the same key material. These form our first type of keyed algorithms and they are referred to as *symmetric* or *secret-key algorithms*. There are three types of algorithms in this category: *block ciphers*, *stream ciphers*, and *message authentication codes (MACs)*. Block ciphers and stream ciphers are encryption primitives while the message authentication code is used for data and data origin authentication. It is worth observing that we can use a block cipher to build both a stream cipher and a message authentication code. In fact we can even build the keyless hash function out of a block cipher and we will explore all of this in Chap. 4.

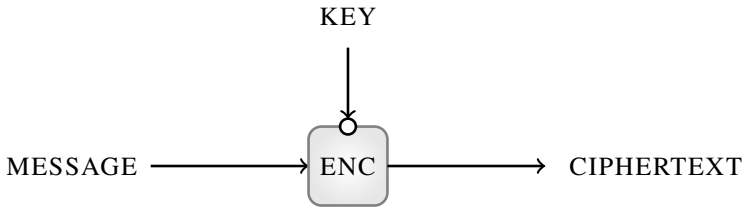
In contrast to the case where the sender and receiver keys are the same, it is possible to devise cryptographic algorithms where the key used by the receiver cannot be computed from that used by the sender. Such algorithms—where the key used for encrypting a message can be made public—are referred to as *asymmetric* or *public-key algorithms*. Different types of algorithms in this category would include *key agreement algorithms*, *public-key encryption algorithms*, and *digital signature algorithms*. This is a fascinating area of cryptography but not one that we will cover in this book.

The symmetric and keyless algorithms tend to be the workhorses of the cryptographic world. As we move from keyless algorithms to symmetric algorithms, and on to asymmetric algorithms, the algorithms at our disposal tend to be slower. An old rule of thumb used to be that a block cipher might be around 50 times faster than an asymmetric algorithm, a stream cipher might be around twice as fast as a block cipher, and a hash function might be faster still at around three times the speed of a block cipher. However, current design trends in all three fields, block ciphers, stream ciphers, and hash functions, are providing more and more exceptions to this crude comparison and it shouldn't be relied upon too closely. However, it does at least help to set the perspective with regard to symmetric and asymmetric cryptography and it illustrates why we leave the bulk data processing—for both encryption and authentication purposes—to the symmetric and keyless algorithms.

1.2 Block Ciphers

As their name implies, block ciphers operate on "blocks" of data. Such blocks are typically 64 or 128 bits in length and they are transformed into blocks of the same size under the action of a secret key. When using the basic block cipher with the same key, two instances of the same input block will give the same output blocks.

The block cipher encrypts a block of *plaintext* or *message* m into a block of *ciphertext* c under the action of a secret key k. This will typically be denoted as $c = \text{ENC}_k(m)$. The exact form of the encryption transformation will be determined by the choice of the block cipher and the value of the key k. The process of encryption is reversed by decryption, which will use the same user-supplied key. This will be denoted $m = \text{DEC}_k(c)$. Throughout the book, the key input to a block cipher will be indicated with a small circle:



A block cipher has two important parameters:

1. the *block size*, which will be denoted by b, and
2. the *key size*, which will be denoted by κ.

For a given key, a b-bit block cipher maps the set $\mathscr{M}$ of 2^b b-bit inputs onto the same set of 2^b outputs:

$$\mathscr{M} = \{\overbrace{0\ldots00}^{b}, \overbrace{0\ldots01}^{b}, \overbrace{0\ldots10}^{b}, \overbrace{0\ldots11}^{b}, \ldots\ldots, \overbrace{1\ldots1}^{b}\}.$$

This is done in such a way that every possible output appears once and only once. The mapping is a permutation of the set of inputs and, as we vary the secret key, we obtain different permutations. Thus a block cipher is a way of generating a family of *permutations* and the family is indexed by a secret key k.

For a secure block cipher we expect no exploitable information about the encryption process to leak. Such information might include information about the choice of key, information about the encryption or decryption of as yet unseen inputs, or information about the permutations generated using different keys.

The block size b determines the space of all possible permutations that a block cipher might conceivably generate. The key size κ determines the number of permutations that are actually generated. To appreciate the (difficult) task of the designer it is worth looking at some of the numbers involved. For a block cipher with key size κ there are 2^κ possible keys and each key specifies a permutation of 2^b inputs. There are $(2^b)!$ different permutations on b-bit input blocks which, by using Stirling's approximation, is approximately $2^{(b-1)2^b}$. For typical values of b and κ a

block cipher will provide only a tiny fraction of all the available permutations. Furthermore, it will do so in a highly structured way. However, we expect a good block cipher to disguise this and, casually speaking, we expect a randomly chosen key to "select" a permutation seemingly at random from among all $2^{(b-1)2^b}$ possibilities. Being even more demanding, we require that keys that are related in some way yield permutations that have no discernible relation between them.

In the chapters to come we will give detailed descriptions of some prominent block ciphers that accomplish this remarkable feat. In the remainder of this chapter, however, we will explore some basic aims of the designer and the adversary.

1.3 Cryptographer and Cryptanalyst

Much of our work in block ciphers is due to the work of Shannon (1916-2001). In particular, the landmark paper *Communication Theory of Secrecy Systems* of 1949 [680] introduced the twin ideas of *confusion* and *diffusion* for practical cipher design. They are still the most widely used principles in block cipher design.

Shannon was working on a mathematical framework for encryption and he was particularly interested in the encryption of natural languages. It is well known that the letter *e* is the most frequently occurring letter in the English language and frequency tables for single letters and digrams or trigrams of letters can be compiled.[1] A cryptanalyst trying to recover some plaintext in natural English has a great deal of side-information specific to the way English is structured. The plaintext source is said to contain *redundancy*.

Historical or classical ciphers [493] employed simple mechanisms to break up the arrangement of the source plaintext. Yet increasingly complex analysis still allowed the cryptanalyst to use natural language redundancy to decrypt the ciphertext. As well as providing a mathematical framework for the process of encryption using statistically based *Information Theory*, Shannon observed that any good encryption algorithm must disguise redundancy in the source plaintext. To quote Shannon;

> Any significant statistics from the encryption algorithm must be of a highly involved and very sensitive type—the redundancy has been both diffused and confused by the mixing transformation [680].

The idea of confusion is "to make the relation between the simple statistics of the ciphertext and the simple description of the key a very complex and involved one", while in the method of diffusion "the statistical structure of the plaintext which leads to its redundancy is dissipated into long range statistics in the cryptogram" [680]. It should be stressed that the properties of confusion and diffusion are not absolute,

[1] The exact distribution will depend on the source. A scientific treatise on zinc may have an inadvertently high number of *z*'s. Similarly the novel *A Void* by George Perec (1938-1982) has a deliberately skewed distribution of letters since the entire text eschews the letter *e* (creating a "void").

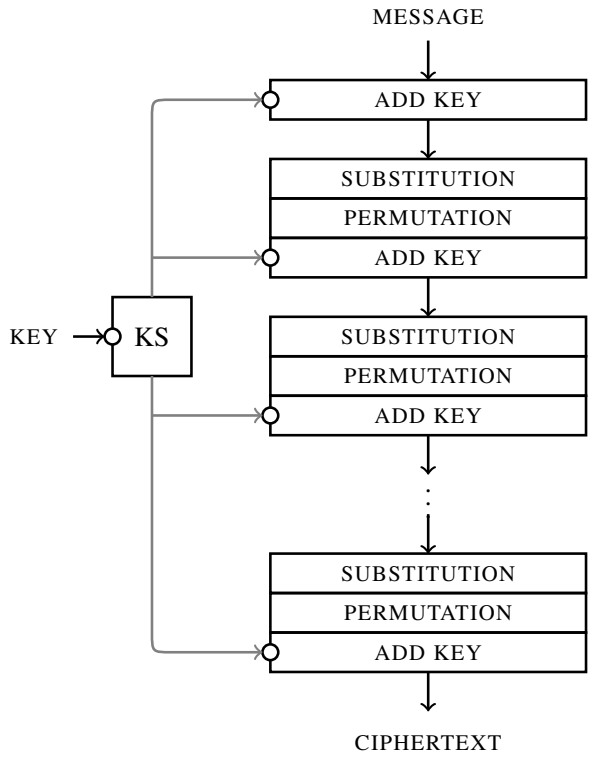


Fig. 1.1 An SP-network. The user-supplied key is processed using a *key schedule* (KS) to derive a set of *round keys*.

quantifiable, concepts. They have been reinterpreted by commentators in different ways and one nice description is given by Massey [469]:

> CONFUSION: The ciphertext statistics should depend on the plaintext statistics in a manner too complicated to be exploited by the cryptanalyst.
>
> DIFFUSION: Each digit of the plaintext and each digit of the secret key should influence many digits of the ciphertext.

Block ciphers are designed to provide sufficient confusion and diffusion. It is the task of the designer to come up with a judicious mix of components that will give a secure (and efficient) block cipher, though it is sometimes difficult to see exactly what contribution each component makes to the explicit goals of confusion and diffusion. However, these concepts remain useful in highlighting the kind of behaviour we expect from a good block cipher.

We thus have some notion of the properties we're striving for. It turns out that the basic operations of *substitution* and *permutation* are particularly important in achieving these goals. Most, if not all, block ciphers will contain some combination of substitution and permutation, though the exact form of the substitution and the permutation may vary greatly.

Substitution is often used as a way to provide confusion within a cipher. Such substitution might be designed around an arithmetic function such as *integer addition* or *integer multiplication*. More typically, substitution is achieved with a suitably designed *lookup table*, *substitution box*, or what we simply refer to as an *S-box*. S-boxes can be carefully designed to have specific security properties and they can be a quick operation in practice. One downside is that S-boxes typically need to be stored, thereby embodying some memory constraints. Provided the boxes are not too large, this is unlikely to cause a significant problem. However, in demanding hardware implementations space can be at a premium [116] while in software implementations interactions between the S-box and on-processor cache can lead to timing variations that potentially key-related information [41, 584].

Permutations are often used to contribute to the good diffusion in a cipher. Very often the permutation is performed at the bit level, by which we mean that individual bits are moved into a new ordering and DES (see Chap. 2) is the obvious example of this approach. However, bit-level permutations have important performance implications. While they are easy to achieve in hardware, where implementation is simply a matter of aligning wires, software is suited to operations that operate on words of data. Manipulating individual bits is not especially natural and this can slow down the performance of the cipher. We will see in a more modern cipher such as the AES (see Chap. 3) that diffusion is provided by byte-level operations with some optimisations exploiting 32-bit words, and this leads to a flexible performance profile.

The mix of substitution and permutation is an important component of most block cipher designs. Indeed, an important class of block ciphers illustrated in Fig. 1.1 is designed using just these operations. Called *substitution/permutation networks* or simply *SP-networks*, these ciphers consist of the repeated application of a carefully chosen substitution, a carefully chosen permutation, and the addition of key material. In Fig. 1.1 we introduce something called the *key schedule*. This is an important feature of block ciphers that rely on several rounds of computation.[2] While the user supplies the encryption key, it is a good design principle to reuse as much of that key material as often as possible throughout the encryption process. It is the role of the key schedule to present a series of *round keys* to each round of encryption and these round keys are computed from the user-supplied encryption key. Designers have different approaches to good key schedules; some key schedules are computationally lightweight whereas others are very complex.

So far we have considered the role of the cryptographer; now we turn our attention to the cryptanalyst. What is it that the attacker is trying to do? In the most extreme case the cryptanalyst will want to recover the user-supplied secret key. However, the attacker may be satisfied with much less. With this in mind it is possible to establish a hierarchy of possible attacks [382]:

1. TOTAL BREAK: The attacker recovers the user-supplied key k.
2. GLOBAL DEDUCTION: The attacker finds an algorithm A that is functionally equivalent to either $\text{ENC}_k(\cdot)$ or $\text{DEC}_k(\cdot)$.

[2] When each round is the same the cipher is sometimes called an *iterated cipher*.

3. LOCAL DEDUCTION: The attacker can generate the message (or ciphertext) corresponding to a previously unseen ciphertext (or message).
4. DISTINGUISHING ALGORITHM: The attacker can effectively distinguish between two black boxes; one contains the block cipher with a randomly chosen encryption key while the other contains a randomly chosen permutation.

An attacker achieving a total break can achieve all the other goals. Indeed these attacks have been ordered so that achieving any given goal automatically achieves those that follow. Taking the converse argument, if an attacker is unable to distinguish between the implementation of a block cipher and a randomly chosen permutation then we have, in some sense, achieved an ideal block cipher. This is the basis for an argument often used within the *provable security* research community where a great deal of work is done on "proving" the security of a construction that uses a block cipher on the assumption that the underlying block cipher offers some "ideal behaviour", *i.e.*, that it cannot be efficiently distinguished from a randomly chosen permutation.

For our analysis we always take it for granted that the cryptanalyst knows the details of the encryption technique. In 1883, Kerckhoffs (1835-1903) laid out six requirements for a usable field cipher [345]. These have been reinterpreted by commentators and the term *Kerckhoffs' Assumption* or *Kerckhoffs' Principle* is now used to refer to the assumption that the cryptanalyst knows every detail of the encryption mechanism except the user-supplied secret key.

We might also assume that the cryptanalyst has access to different types of data. Indeed, as we look at attacks in more detail it will become clear that we need to specify exactly the kind of data required in an attack. Not all kinds of data are equally useful, nor equally available, and attacks can be classified according to the type of data that they require:

1. CIPHERTEXT ONLY: The attacker intercepts the ciphertext generated by the encryption algorithm. Here the attacker will rely on some knowledge of the plaintext source; *i.e.*, some form of redundancy.
2. KNOWN PLAINTEXT/MESSAGE: The attacker is able to intercept a set of n ciphertexts $c_0 \dots c_{n-1}$ corresponding to some known messages $m_0 \dots m_{n-1}$ with $c_i = \text{ENC}_k(m_i)$ for $0 \leq i \leq n-1$.
3. CHOSEN PLAINTEXT/MESSAGE: The attacker is able to request the encryption of a set of n messages $m_0 \dots m_{n-1}$ of the attackers' choosing[3] and intercepts the ciphertexts $c_i = \text{ENC}_k(m_i)$ for $0 \leq i \leq n-1$.
4. ADAPTIVELY CHOSEN PLAINTEXT/MESSAGE: The attacker obtains the encryption of new chosen messages m_i for $i \geq n$ in an interactive way, perhaps after seeing an original pool of n chosen message/ciphertext pairs $c_i = \text{ENC}_k(m_i)$ for $0 \leq i \leq n-1$.

[3] This might appear unlikely but there are situations where this is a real possibility. During the Second World War the British RAF would drop mines at specific chosen grid references (a process referred to as "gardening") so that their opponents, on discovering the mines, would generate warning messages exploited by the cryptanalysts at Bletchley Park [296].

5. CHOSEN AND ADAPTIVELY CHOSEN CIPHERTEXT: The attacker recovers the decryption of ciphertexts of his own choosing. As in the case of a chosen message attack, there is an adaptive version.

With each escalation in the type of data available, the attacker has more control over the analysis of the block cipher and can devise increasingly sophisticated attacks. However, at the same time collecting data of a given type becomes more demanding as we move down the list. Many of the attacks we consider will be differentiated according to how much data is required as well as the type of data required.

The success of different types of cryptanalysis will be measured according to the resources they consume. We have already seen that the amount, and type, of data is important. We can add some additional resources that are likely to be of interest below:

1. TIME: The time, or work effort, required to mount the attack is usually the first thing analysts and commentators consider in a new attack. Sometimes—and therefore potentially erroneously—it is the only requirement considered.
2. MEMORY: The amount of memory, or storage, for an attack is very important. Sometimes the amount of memory is so great that it creates an insurmountable bottleneck and the attack remains impractical. One frequently cited motto is that "time is cheaper than memory" [288]. While it might not provide an eternal truth, it nicely captures the idea that the time to perform, say, 2^{40} encryption operations is easier to accumulate than the memory to store the 2^{40} results.
3. DATA: We have already seen that the type of data for an attack is important, but so is the amount of data. For instance, an attack might not require much time but it might require an enormous amount of data. If the time required to generate the data far exceeds normal usage patterns then the practical impact of the attack is limited.

When considering a cryptographic scheme in isolation, the data types and the resources listed above are typically the most appropriate way to quantify a cryptanalytic attack. However, when we turn to cryptographic implementation and deployment, a whole range of new attacks that can be more powerful than classical cryptanalysis become vital. These are attacks that allow the attacker to exploit the leakage of physical information, for instance, the encryption time [411] or the power consumption [412], during the implementation of an algorithm. This area is termed *side-channel analysis*, and the design of attacks and countermeasures is a vast and constantly evolving field. Regrettably, however, including such work would take us out of the scope we have already established.

Instead we turn our attention to the simplest and most widely applicable method of cryptanalysis, that of guessing the secret encryption key.

Table 1.1 A broad headline assessment of the security offered by different key lengths, in the absence of any cryptanalysis weakness.

κ *(bits)*	*search time (operations)*	*Current Status (2010)*	
40	2^{40}	easy to break	no security
64	2^{64}	practical to break	poor security
80	2^{80}	not currently feasible	reasonable security
128	2^{128}	very strong	excellent security
256	2^{256}	exceptionally strong	astronomical levels of security

1.4 Security

To use a block cipher we must choose a secret key k. The easiest attack an adversary can mount is to simply try and guess the value of the key being used. If the key is κ bits long there are 2^{κ} alternatives. The probability of correctly guessing the key at the first attempt is $2^{-\kappa}$. Adding an additional bit to the length of the key halves the probability that the key is correctly guessed.

A more systematic approach is to exhaustively search for the secret key. No matter how clever or secure the algorithm, this type of attack is always available to the cryptanalyst and the only way to protect against it is to have a sufficiently large key. The time required to exhaust all possible keys (the *keyspace*) is proportional to the time required to perform 2^{κ} encryption operations. Having to exhaust the entire keyspace before finding the correct key would be very unlucky, just as being correct on the first guess would be very lucky. So often we refer to the *expected time* to recover a κ-bit key which is $2^{\kappa-1}$ operations.

In translating this to a physical time, testing a key depends on the time to encrypt a block of message and the time to run the key schedule. Recall that for a multi-round block cipher we need to extract a set of round keys from the user-supplied key. Normally the cost of running the key schedule is not so significant. It occurs once at the start of encryption and any time penalty is spread or amortised over the whole encryption process. However, during exhaustive search we are changing keys often and the cost of the key schedule can be significant. For example, it is estimated that testing an RC5 key takes four times longer than a DES key [650].

In Chap. 5 we will consider some sophisticated variants of exhaustive search. These use memory to reduce the time required to find the secret key. But in its raw form exhaustive key search has no memory requirements and has very light data requirements. To identify the correct key we would rely on redundancy in the message source so that we can recognise when the decrypted plaintext is correct. Of course, the best way to do this would be with a known message.

Depending on the block and key size, it is possible that several keys map one particular message to the same ciphertext. For a good b-bit block cipher the probability that a key maps a given message to a specific ciphertext should be 2^{-b}. Considering the entire keyspace we would therefore expect $2^{\kappa} \times 2^{-b}$ keys to provide a match for a single message/ciphertext pair. Depending on the sizes of κ and b we might therefore need additional message/ciphertext pairs to eliminate any false alarms from our exhaustive key search.

It is interesting to note that changing encryption keys does not necessarily provide much additional protection against exhaustive key search. To illustrate, we suppose that κ is not that large and that we can exhaustively test the keyspace in a year. This means that in the course of any single day we can search $\frac{1}{365} \approx 0.3\%$ of all possible keys. Suppose now that the key is changed every day. Then the probability that we will recover the key that is used on a given day is around 0.3%. If we were to search over the course of an entire year then we would expect to be successful at least once. While the attacker cannot predict which of the 365 keys he will recover during the year, he should get one of them. Thus there is still some risk even though keys are being changed frequently. It is however slightly better than the case of a key that remains unchanged. In this case the attacker would be guaranteed to be successful in recovering an active key during that year of searching.

So how do we know when a key is large enough? Much will depend on the risks facing the application and no definitive answer can be given. However, it seems that there is reasonable consensus in the cryptographic community and several benchmark key sizes have been established; see Table 1.1. To provide a physical comparison to these enormous numbers, the universe is less than 2^{80} microseconds old and it is estimated to contain around 2^{250} protons.

1.5 Summary

During our opening discussion on exhaustive search we have identified a *necessary* condition for the practical security of a block cipher, namely that the encryption key k be sufficiently long. Note, however, that this is not a *sufficient* condition and the field of cryptography is littered with poor algorithms whose only claim to security is a large key!

Indeed it might be tempting to ensure that every block cipher—even those of high-quality design—have an enormously long secret key. However, one problem in the deployment of cryptography is that of generating and managing secret keys. We really don't want to have to deal with more key material in a system than we need to. There are also significant concerns about how one might go about designing a cryptosystem that uses an enormously long key without any attendant compromise in performance. But perhaps most importantly from the authors' perspective, it is just not very elegant.

The rest of the book, therefore, is intended to help us identify some block ciphers that achieved this careful balancing act. But what is a *good* block cipher? In our

opinion a good block cipher is one for which the best attack is an exhaustive search. Such a cipher performs according to the designers' intentions. However, a good block cipher might not necessarily be a practically secure block cipher; for instance, the key length could be shorter than we would like and exhaustive search could therefore pose more of a threat than we would like. In the next chapter we will encounter such a cipher.

Chapter 3
AES

The influence of DES on the development of secret-key cryptography is immense. Its success inspired many closely related designs, though improving on DES was always a rather lofty ambition. Indeed, very few of these DES-relatives offered any serious advantages and many ended up being weaker than the original.

The development of the Advanced Encryption Standard (AES) allowed the cryptographic community to take a major step forward in block cipher design. Just as DES trained a generation of cryptographers, the AES process provided a massive spur to the design, analysis, and implementation of block ciphers. Ironically, the establishment and standardisation of the AES tended to dampen, at least for a while, any new research on block cipher designs. While some researchers continued to analyse the AES, there was little real purpose (or indeed desire) to design alternative block ciphers. The situation has now changed somewhat and new designs for new applications are now being proposed. However, unless there is a catastrophic weakness in the AES, this is the cipher that will be used almost everywhere for the next 20 to 30 years.

Before we describe today's most important cipher, it is worth reviewing the process by which the block cipher *Rijndael* [185] was finally chosen as the AES on October 2, 2000. As we will see in Chap. 9, the 1990s witnessed an explosion in block cipher proposals. Many of these were never taken up, but a fair number were deployed in products and some are in use today. Several of these ciphers contained interesting architectural features that later found their way into other block cipher designs as well as into some of the ciphers in the final round of the AES competition.

Between January 1997 and November 2001, when FIPS 197 was finally published [551], the National Institute of Standards and Technology (NIST) coordinated the search for a DES replacement. Designers had a clean slate in terms of putting together a cipher matching the NIST-specified requirements, and these requirements were few but clear:

1. The cipher should be a single block cipher.
2. The cipher should be available royalty-free worldwide.
3. The cipher should have a public and flexible design.
4. The cipher should offer the security of two-key triple-DES as a minimum.

L.R. Knudsen and M.J.B. Robshaw, *The Block Cipher Companion*, Information Security and Cryptography, DOI 10.1007/978-3-642-17342-4_3, © Springer-Verlag Berlin Heidelberg 2011

Some specific parameters were set for the new AES. Among them was the stipulation that a block size of 128 bits should be supported and that any proposal should be able to support at least three key sizes of $\kappa = 128$, 192, and 256 bits.[1]

The AES process was run as an open competition and there were entries from all over the world. Out of 21 entries, 15 were accepted for first round consideration. This first round of evaluation lasted around a year and depended for the most part on input from the cryptographic community. At the end of the first round, five finalists were chosen for second round consideration and, after DES, they are perhaps the most scrutinized block ciphers available today. All finalists had positive and negative attributes, all had distinctive (and elegant) design attributes, and yet they were radically different. The best way to get an appreciation of the full range of issues involved in the AES process—and the enormous amount of work undertaken by NIST, the submitters, and analysts alike—is to look at the final NIST report [550]. The AES winner was the block cipher Rijndael proposed by Joan Daemen and Vincent Rijmen. An elegant cipher, Rijndael offered the most appealing performance profile of the AES finalists and this made it a popular choice among implementors.

In the coming sections we will describe the AES/Rijndael. First though, we need to decide how we are going to refer to the algorithm since there is a subtle difference between them. The submission Rijndael was a more flexible cipher than the final standardised AES since Rijndael accommodates multiple block lengths and multiple key lengths. However, the AES is explicitly restricted to a block length of 128 bits and a key length of 128, 192, or 256 bits. Since there are good technical reasons to have a standardised block cipher with a block length of 256 bits (see Chap. 4), some view this as a missed opportunity[2]. Others, however, focus on the issue of interoperability and stress that it is one thing to have applications negotiate a key length but another, more complicated issue to have applications that support multiple block lengths.

3.1 AES Description

The AES is an SP-network and so there will be clearly defined layers of S-box substitution and diffusion. Perhaps the most striking feature of the AES is the very regular byte structure that is maintained throughout the cipher. Every operation takes place on bytes and this makes it very versatile for implementation. Indeed, encryption with the AES is best described with reference to a square array of bytes. A second notable feature of the AES is that whenever a component can be reused, it is reused, and whenever alternatives are available, the simplest and most efficient option is taken. The net result is a cipher that is very attractive to implementors.

It might not be immediately obvious from the description that follows, but the security of the AES depends on viewing a single byte of data in two different ways.

[1] Initial calls from NIST also discussed the requirement for block sizes $b = 128$, 192, and 256 bits.

[2] In fact FIPS 197 [551] mentions (Sect. 6.3) the possibility of supporting different block sizes in future versions of the standard.

The first way is to view a byte as a string of eight bits. This is the representation of a byte that we have used most of the time so far. For those who prefer a mathematical formalization, we are regarding eight bits as an eight-dimensional vector over GF(2). A second way to treat a byte is to view it as representing an element of the field GF(2^8) of 256 elements. Operations that are easy to analyse in one representation can be difficult to cope with (as a cryptanalyst) in the other representation. By mixing operations that combine bytes in different ways a very strong cipher can be derived. More background on this and related issues is provided in [159].

3.1.1 Arithmetic in GF(2^n)

Entries in GF(2^n) can be expressed as n-bit strings. Equivalently, they can be viewed as polynomials of degree up to $n-1$. In this way all 2^n members of GF(2^n) can be written as a polynomial

$$c_{n-1}X^{n-1} + c_{n-2}X^{n-2} + \ldots c_2X^2 + c_1X + c_0$$

where c_0 to c_{n-1} are coefficients that take the values 0 or 1.

The arithmetic defined over GF(2^n) allows us to both add and multiply these values. Adding elements is straightforward. Two polynomials of degree $n-1$ are added termwise, with the resultant coefficients being computed modulo 2. Thus

$$\begin{aligned}
&(c_{n-1}X^{n-1} + c_{n-2}X^{n-2} + \ldots c_2X^2 + c_1X + c_0)\\
+\ &(d_{n-1}X^{n-1} + d_{n-2}X^{n-2} + \ldots d_2X^2 + d_1X + d_0)\\
=\ &(c_{n-1} \oplus d_{n-1})X^{n-1} + (c_{n-2} \oplus d_{n-2})X^{n-2} + \ldots\\
&+(c_2 \oplus d_2)X^2 + (c_1 \oplus d_1)X + (c_0 \oplus d_0).
\end{aligned}$$

For multiplication we need to specify the *irreducible polynomial* of degree n that defines the representation of the field GF(2^n). We might denote this polynomial by R_p and we have

$$R_p = X^n + y_{n-1}X^{n-1} + y_{n-2}X^{n-2} + \ldots y_2X^2 + y_1X + y_0.$$

Now when we multiply two entries in GF(2^n) we can view this as multiplying two polynomials of degree $n-1$. Since the coefficients are all $\{0,1\}$ the polynomial that results will have $\{0,1\}$ coefficients. If the degree of this product is $n-1$ or less, then we are done and we have our entry in GF(2^n). Otherwise, the product contains some nonzero coefficients for the terms X^n or higher. If the largest power of X that appears in the intermediate product is the term X^{n+i} for some $i \geq 0$, then we add the polynomial $R_p \times X^i$ to the intermediate result. This will eliminate the leading term X^{n+i}, while likely changing the values of the lower coefficients. We then look for the next greatest power of X that remains and repeat the process. This is, in effect, a method to reduce the polynomial product modulo R_p.

For the purposes of the AES we will be operating on bytes and so we have $n = 8$. The field representation used for the AES is defined by what is now referred to as the *Rijndael polynomial* where

$$R_p = X^8 + X^4 + X^3 + X + 1.$$

3.1.2 Encryption with the AES

The AES is defined for 128-bit blocks and keys of length 128, 192, and 256 bits. We will refer to these three variants as AES-128, AES-192, and AES-256. The essential differences between these three variants are the number of rounds used for encryption and some slight changes to the key schedule [551].

To begin, the 128-bit input can be written in bits as $b_0 b_1 \dots b_{127}$. These are divided into 16 bytes $B_0 B_1 \dots B_{15}$ where we set $B_i = b_{8i} b_{8i+1} \dots b_{8i+7}$. We know that this byte can be viewed as an element of $\mathrm{GF}(2^8)$ and so we can equivalently write byte B_i, for $0 \leq i \leq 15$, as a polynomial of degree 7:

$$b_{8i}X^7 + b_{8i+1}X^6 + b_{8i+2}X^5 + b_{8i+3}X^4 + b_{8i+4}X^3 + b_{8i+5}X^2 + b_{8i+6}X + b_{8i+7}.$$

Conceptually, we then place these bytes in a square array as follows:

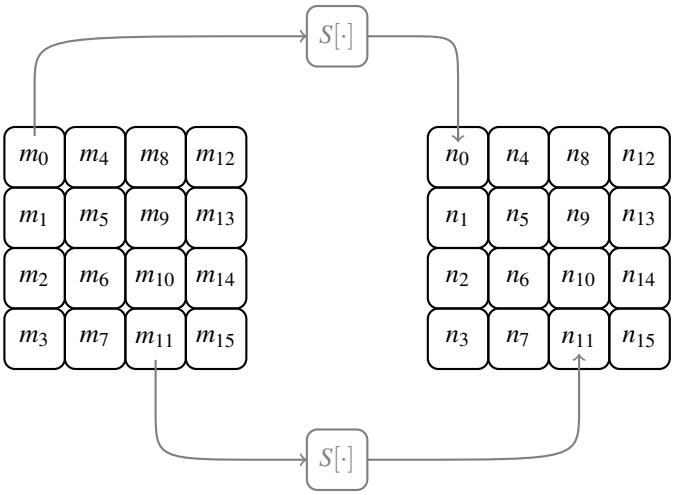
B_0	B_4	B_8	B_{12}
B_1	B_5	B_9	B_{13}
B_2	B_6	B_{10}	B_{14}
B_3	B_7	B_{11}	B_{15}

The AES round function uses the following operations: `SubBytes` transforms each byte of the array using a nonlinear substitution box; `ShiftRows` ensures that each row of the array is moved by a different number of byte positions; `MixColumns` mixes each column of the array while `AddKey` mixes each byte of the array with a byte of subkey material. We will now look at each of these operations in turn.

3.1.2.1 `SubBytes`

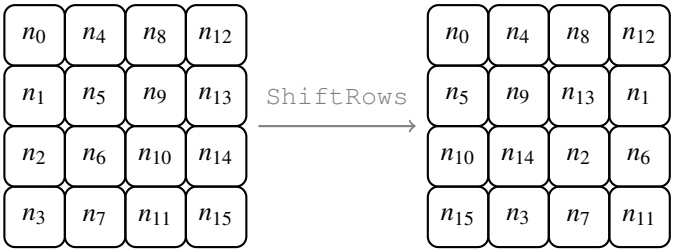
Each byte of the array is transformed by the AES S-box. This takes in an eight-bit quantity and gives an eight-bit quantity as output. Only one S-box is used throughout the cipher and, as we will see, it has been carefully constructed. So an array $m_0, \dots, m_{15}$ is transformed into an array $n_0, \dots, n_{15}$ using the following identities (which we illustrate for $i = 0$ and $i = 11$):

$$n_i = S[m_i] \text{ for } 0 \leq i \leq 15.$$



3.1.2.2 `ShiftRows`

Each row of the array is moved to the left by a different number of byte positions. More precisely, row i is moved by i byte positions for $0 \leq i \leq 3$.

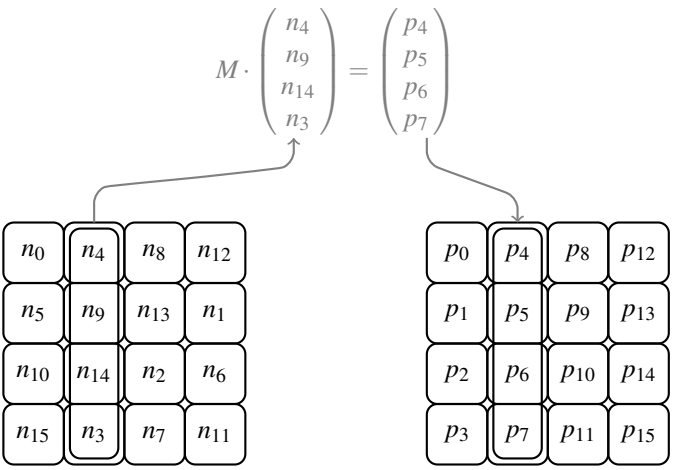


3.1.2.3 `MixColumns`

Each column of the array is mixed together. To do this, we view the column as a (4×1) column vector of entries in GF(2^8) and we pre-multiply this column vector by the (4×4) GF(2^8)-matrix M where

$$M = \begin{pmatrix} \texttt{02 03 01 01} \\ \texttt{01 02 03 01} \\ \texttt{01 01 02 03} \\ \texttt{03 01 01 02} \end{pmatrix}.$$

The output will be a (4×1) column vector of entries in GF(2^8) which replaces the column being processed.



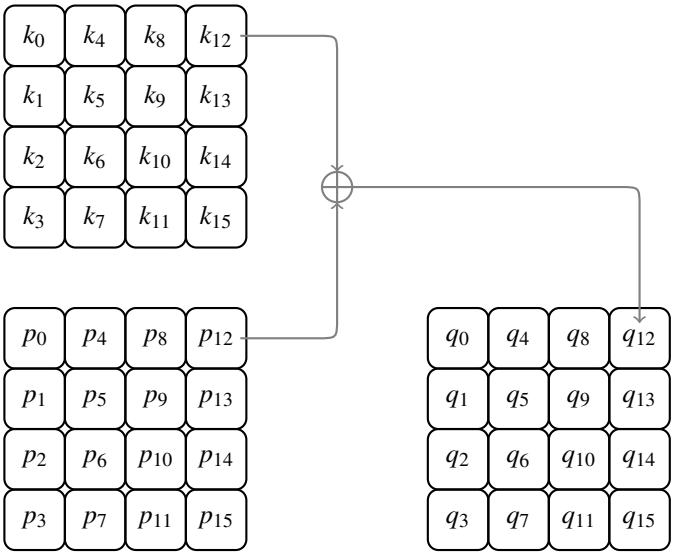
The product of two polynomials for the `MixColumns` operation can be easily described as regular multiplication of polynomials, but with any term of degree greater than 7 being reduced modulo the Rijndael polynomial R_p; see Sect. 3.1.1. Thus, as an example, the product of the two bytes `02` and `a3` can be computed as

$$\begin{aligned} \texttt{02} \times \texttt{a3} &= \\ X \times (X^7 + X^5 + X + 1) &= X^8 + X^6 + X^2 + X \\ &= R_p + (X^4 + X^3 + X + 1) + (X^6 + X^2 + X) \\ &= X^6 + X^4 + X^3 + X^2 + 1 \\ &= \texttt{5d}. \end{aligned}$$

3.1.2.4 `AddRoundKey`

At each round, the array of bytes (the *state array*) is combined with a similarly sized array of subkey material. This is derived from the user-supplied key using a key schedule, that will be described in Sect. 3.1.4.

If we denote the current state of the byte array by $p_0 \dots p_{15}$ and the subkey array as $k_0 \dots k_{15}$ then the result of `AddRoundKey` is to combine these arrays in a bytewise fashion to give an array $q_0 \dots q_{15}$ where $q_i = p_i \oplus k_i$ for $0 \leq i \leq 15$. We illustrate this for $i = 12$.



3.1.2.5 Overall Structure for Encryption

We have seen the form of a basic AES round. There are a couple of additional details to consider. The first is that encryption begins with an iteration of `AddRoundKey` where the round key in question is part of the user-supplied key material. We will see this more when we consider the key schedule in Sect. 3.1.4. The second detail is that the final round of the AES does not use the `MixColumns` operation and this allows us to write the decryption process in a way that is very similar to encryption. So, in summary, we have the following form to encryption over n rounds of the AES. This is also illustrated in Fig. 3.4 on page 49. For AES-128 we have $n = 10$ rounds, for AES-192 we have $n = 12$ rounds, and for AES-256 we must use $n = 14$ rounds.

AES Encryption over n Rounds	
`AddRoundKey`	pre-whitening
`SubBytes`	↑
`ShiftRows`	for $n-1$
`MixColumns`	rounds
`AddRoundKey`	↓
`SubBytes`	
`ShiftRows`	final round
`AddRoundKey`	

3.1.3 Decryption with the AES

For each of the operations in a round, it is straightforward to define the corresponding inverse operation.

Encryption	**Decryption**
`SubBytes`: use S-box $S[\cdot]$	`InvSubBytes`: use S-box $S^{-1}[\cdot]$
`ShiftRows`: rotate to the left	`InvShiftRows`: rotate to the right
`MixColumns`: multiply by M	`InvMixColumns`: multiply by M^{-1}
`AddRoundKey`	`AddRoundKey`

To decrypt, just use these operations in the correct order so as to undo the encryption process, making sure to reverse the order of the round keys. Noting that the operation `AddRoundKey` is self-inverse (*i.e.*, an *involution*), we have the following form to decrypt over n rounds.

AES Decryption over n Rounds	
`AddRoundKey`	
`InvShiftRows`	undo final round
`InvSubBytes`	
`AddRoundKey`	↑
`InvMixColumns`	for $n-1$
`InvShiftRows`	rounds
`InvSubBytes`	↓
`AddRoundKey`	undo pre-whitening

Since the operations `InvSubBytes` and `InvShiftRows` operate in a byte-wise fashion they commute with one another. Further, if we make some slight adjustments to the decryption key schedule we can reverse the order of the `AddRoundKey` and `InvMixColumns` operations. This leads us to the following description of the AES decryption process which emphasises a similarity between the forms of encryption and decryption.

Alternative form to AES Decryption over n Rounds	
`AddRoundKey`	pre-whitening
`InvSubBytes`	↑
`InvShiftRows`	for $n-1$
`InvMixColumns`	rounds
`AddRoundKey`	↓
`InvSubBytes`	
`InvShiftRows`	final round
`AddRoundKey`	

For completeness, $S[\cdot]$ and $S^{-1}[\cdot]$ are given in Table 3.1 while the matrices M and M^{-1} are given by

Table 3.1 The AES substitution and inverse substitution boxes $S[\cdot]$ and $S^{-1}[\cdot]$. The value of $S[\texttt{ab}]$ is given by the entry in row a and column b.

$S[\cdot]$

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

$S^{-1}[\cdot]$

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

$$M = \begin{pmatrix} \texttt{02} & \texttt{03} & \texttt{01} & \texttt{01} \\ \texttt{01} & \texttt{02} & \texttt{03} & \texttt{01} \\ \texttt{01} & \texttt{01} & \texttt{02} & \texttt{03} \\ \texttt{03} & \texttt{01} & \texttt{01} & \texttt{02} \end{pmatrix} \quad \text{and} \quad M^{-1} = \begin{pmatrix} \texttt{0e} & \texttt{0b} & \texttt{0d} & \texttt{09} \\ \texttt{09} & \texttt{0e} & \texttt{0b} & \texttt{0d} \\ \texttt{0d} & \texttt{09} & \texttt{0e} & \texttt{0b} \\ \texttt{0b} & \texttt{0d} & \texttt{09} & \texttt{0e} \end{pmatrix}.$$

3.1.4 AES Key Schedule

The key schedule for the AES is computationally lightweight and efficient in terms of memory and performance. It takes as input a user-supplied key of 16, 24, or 32 bytes and returns what is termed an `ExpandedKey` of 16×11, 16×13, and 16×15 bytes respectively. Descriptions of the generation of `ExpandedKey` can be found in [189, 551] while we illustrate the process in Figs. 3.1, 3.2, and 3.3.

For the most part we will be able to describe the key schedule in terms of 32-bit words or four-byte columns of the state array of the AES. Since there are four 32-bit words in each round subkey in the AES, which together give a 128-bit round key, we will denote these words at round i, for $0 \leq i \leq n$, as $k_{(i,0)} \dots k_{(i,3)}$, and the subkey for round i is given by $k_{(i,0)} \| k_{(i,1)} \| k_{(i,2)} \| k_{(i,3)}$. So $k_{(i,0)}$ is used as the first column of the round-key array, and so on through to $k_{(i,3)}$, which is used as the last column. For $i = 0$ the first subkey is derived directly from the user-supplied key. For AES-128 we have $n = 10$, for AES-192 $n = 12$, and for AES-256 $n = 14$.

The AES key schedule works in an iterative manner and at each iteration the next t words of key material are computed as a function of the current t words where $t = \frac{\kappa}{32}$ for AES-κ. Thus, for AES-128 we have $t = 4$, for AES-192 we have $t = 6$, while for AES-256 $t = 8$. In all cases, as we have already mentioned, the first t columns of the AES round keys are derived directly from the user-supplied key. For AES-128 this is straightforward and the round key $k_{(0,0)} \| k_{(0,1)} \| k_{(0,2)} \| k_{(0,3)}$ in the first `AddRoundKey` operation is identical to the user-supplied key. For AES-192, four columns of the user-supplied key are used in the first round subkey $k_{(0,0)} \| k_{(0,1)} \| k_{(0,2)} \| k_{(0,3)}$ while the remaining two words form the first two columns $k_{(1,0)}$ and $k_{(1,1)}$ of the second round subkey. For AES-256, the entirety of both the first two round keys are provided by the user-supplied key.

The subkeys for later rounds are derived in different ways depending on the length of the user-supplied key. For AES-128 the subkey $k_{(i,0)} \| k_{(i,1)} \| k_{(i,2)} \| k_{(i,3)}$ is derived from the subkey in the previous round $k_{(i-1,0)} \| k_{(i-1,1)} \| k_{(i-1,2)} \| k_{(i-1,3)}$. At each round there is a simple operation on one 32-bit word (viewed as a set of four bytes). This consists of a byte rotation, an S-box operation (using the same S-box as for encryption), and the addition of an iteration constant r_i, for $i \geq 1$, which is given by $r_i = 02^{i-1}$ in $\mathrm{GF}(2^8)$.

For AES-192 the subkey generation is very similar. Since the user-supplied key is larger than the state of the AES, we find for $1 \leq i \leq 11$ that two successive values of the key registers, each t words long, hold three successive round keys, see Fig. 3.2.

For AES-256 the subkey generation is a bit more involved. It still uses all the same basic operations as those in the key schedules for AES-128 and AES-192, but there is an additional set of operations included. Since the key registers now contain eights words, two successive values provide four successive round keys. For this reason, each iteration of the key schedule for 256-bit keys involves a bit more computation, see Fig. 3.3.

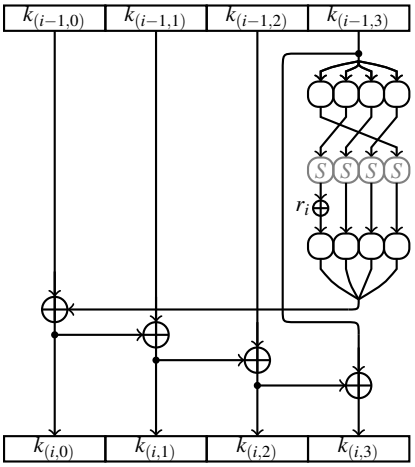


Fig. 3.1 A diagrammatic representation of the key schedule for AES-128. At times the 32-bit words are treated as four bytes, as indicated. The constant r_i for $i \geq 1$ is given by $r_i = 02^{i-1}$ in GF(2^8).

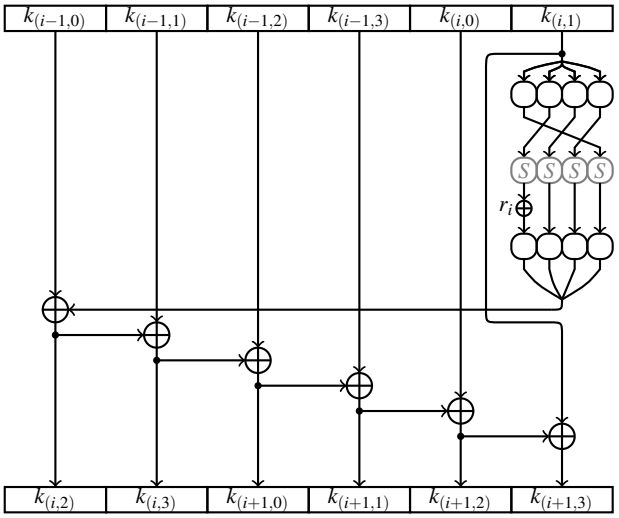


Fig. 3.2 A diagrammatic representation of the key schedule for AES-192. At times the 32-bit words are treated as four bytes, as indicated. The constant r_i for $i \geq 1$ is given by $r_i = 02^{i-1}$ in GF(2^8).

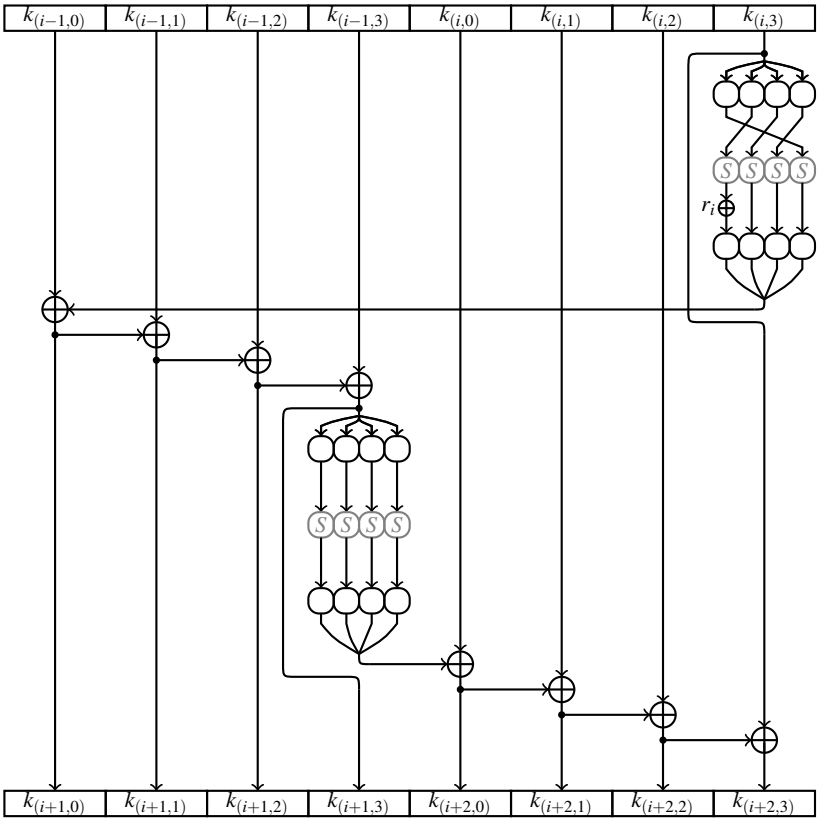


Fig. 3.3 A diagrammatic representation of the key schedule for AES-256. At times the 32-bit words are treated as four bytes, as indicated. The constant r_i for $i \geq 1$ is given by $r_i = 02^{i-1}$ in GF(2^8).

Table 3.2 For the AES test vector given in [551], message `3243f6a8 885a308d 313198a2 e0370734` is encrypted to give the ciphertext `3925841d 02dc09fb dc118597 196a0b32` under the action of the 128-bit user-supplied key `2b7e1516 28aed2a6 abf71588 09cf4f3c`. For this encryption, the ten round keys of 128 bits and the ten intermediate round inputs are as follows.

round	*round input*
1	`193de3bea0f4e22b9ac68d2ae9f84808`
2	`a49c7ff2689f352b6b5bea43026a5049`
3	`aa8f5f0361dde3ef82d24ad26832469a`
4	`486c4eee671d9d0d4de3b138d65f58e7`
5	`e0927fe8c86363c0d9b1355085b8be01`
6	`f1006f55c1924cef7cc88b325db5d50c`
7	`260e2e173d41b77de86472a9fdd28b25`
8	`5a4142b11949dc1fa3e019657a8c040c`
9	`ea835cf00445332d655d98ad8596b0c5`
10	`eb40f21e592e38848ba113e71bc342d2`

round	*round key*
1	`a0fafe1788542cb123a339392a6C7605`
2	`f2c295f27a96b9435935807a7359f67f`
3	`3d80477d4716fe3e1e237e446d7a883b`
4	`ef44a541a8525b7fb671253bdb0bad00`
5	`d4d1c6f87c839d87caf2b8bc11f915bc`
6	`6d88a37a110b3efddbf98641ca0093fd`
7	`4e54f70e5f5fc9f384a64fb24ea6dc4f`
8	`ead27321b58dbad2312bf5607f8d292f`
9	`ac7766f319fadc2128d12941575c006e`
10	`d014f9a8c9ee2589e13f0cc8b6630ca6`

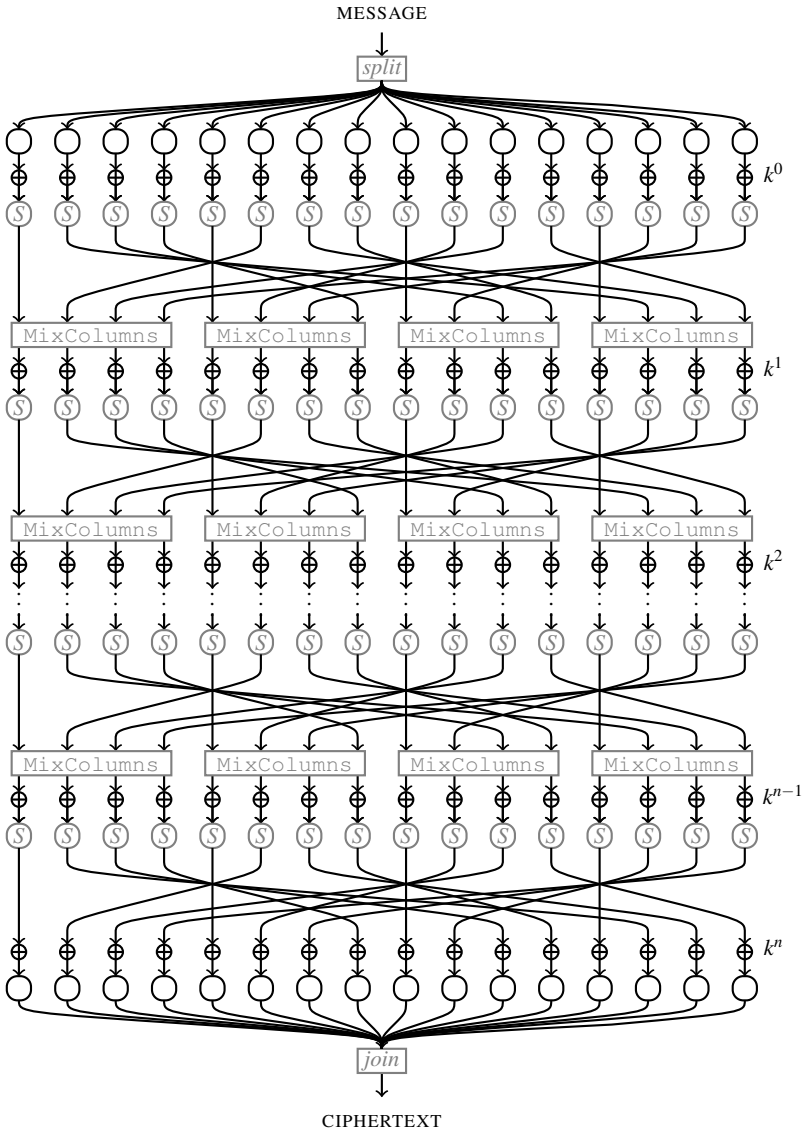


Fig. 3.4 An overview of encryption using the AES. The user-supplied key k is processed using a *key schedule* (*KS*) to derive a set of $n+1$ *round keys* $k^0 \ldots k^n$.

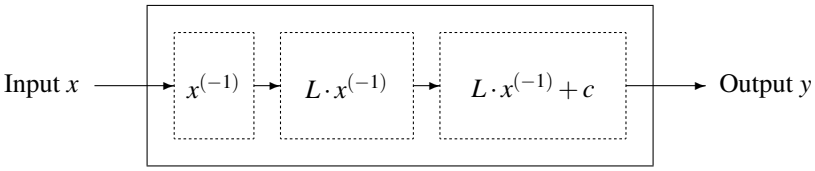
3.1.5 AES Design Features

It is clear that the AES is a very different block cipher from DES. However, there are some similarities in the mix of substitution and permutation components. Both have been very carefully designed and, as we will see in Sect. 3.2, this is particularly the case for the AES when providing resistance to differential and linear cryptanalysis. Just as with DES, the designers chose S-boxes that would provide little advantage to the cryptanalyst and then designed the diffusive components of the cipher to maximise the number of S-boxes a cryptanalyst would have to deal with in some attack. The major difference between DES and the AES lies in the fact that a highly structured approach has been used when building components of the AES.

3.1.5.1 AES Substitution

The AES S-box is given in Table 3.1 where it might not be clear that it has any particular form. Nevertheless, it has been constructed out of three operations.

- Inversion of the input byte x when viewed as an element of $\mathrm{GF}(2^8)$. In reality, this inversion is slightly modified since inversion of 0 would not ordinarily be defined. For the AES we set $(0)^{-1} = 0$ and this modified inversion operation is sometimes written as $x^{(-1)}$.
- Multiplication of a byte by an (8×8) GF(2)-matrix L when the byte is viewed as an (8×1) column vector over GF(2).
- Addition of a constant c to a byte when viewed as an (8×1) column vector over GF(2).



The design rationale is given in a variety of documents [185, 189] and can be summarised as follows.

- Inversion over $\mathrm{GF}(2^8)$ gives good localised resistance to differential and linear cryptanalysis. While it might not mean much until we get to Chaps. 6 and 7, the maximum probability for a difference propagation across the inversion operation is 2^{-6} while the maximum bitwise correlation between the input to and the output from the inversion operation is 2^{-3} [185]. Together with the carefully designed diffusion layer, this is going to give excellent security against these attacks. Many of the theoretical foundations for using such algebraic operations within a block cipher construction were established in [568, 564, 569].
- On its own, the inversion operation is algebraically simple. For instance, even though we know that the bitwise difference does not propagate well across the

$\mathrm{GF}(2^8)$-inversion operation we do know that if $y = x^{(-1)}$ for some $x, y \in \mathrm{GF}(2^8)$ then $yx = 1$ except when $x = y = 0$. To hinder such properties being exploited by an adversary, the designers of the AES immediately follow the inversion operation with a bitwise operation, namely, a transformation of the byte by a $\mathrm{GF}(2)$-matrix that is typically denoted by L. This has no adverse effect on the differential and linear properties of the (modified) inversion operation and yet should provide a barrier to viewing the S-box entirely as an operation over $\mathrm{GF}(2)$ or $\mathrm{GF}(2^8)$. It is in the design of the S-box that we see the conflict between operations in $\mathrm{GF}(2^8)$ and $\mathrm{GF}(2)$ most pronounced. The $\mathrm{GF}(2)$-matrix L is given by[3]

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

- The final operation in the S-box is the addition of a constant c to the byte value that results from the $\mathrm{GF}(2)$-matrix multiplication. The value of the constant is set to $\texttt{01100011} = \texttt{63}_{\texttt{x}}$. Why is there a constant? The designers of the AES expressed a concern that without this addition, a zero input to the S-box would provide a zero output. Thus we would have a fixed point. It is unclear how important this would be for security in the overall cipher, but it was felt that it would be better if all fixed points (and any anti-fixed points for which $S[a] = \overline{a}$) were removed [185].

In Sect. 3.2 we will consider the implications of this S-box design. With regards to established attacks such as differential and linear cryptanalysis it is particularly successful. With some other forms of analysis, however, we will see that this construction opens up some interesting opportunities for the cryptanalyst.

3.1.5.2 AES diffusion

Diffusion within the AES is provided by the `ShiftRows` and `MixColumns` operation. Clearly, the two operations complement each other; `ShiftRows` ensures that any pattern or relationship between elements within the same column is disrupted, while `MixColumns` ensures not only that relationships within a column are difficult to track, but also that elements in the same row will be modified differently. Of course, this is the AES, and so while `ShiftRows` is a fairly routine operation, the designers have designed `MixColumns` so that it functions in a particularly useful

[3] Note that a different bit ordering means that the specification of L in FIPS 197 [551] differs from that provided in *The Design of Rijndael* [189].

way. Recall that `MixColumns` can be described as pre-multiplication of a (4×1) GF(2^8)-column vector by a (4×4) GF(2^8)-matrix M given by

$$M = \begin{pmatrix} \texttt{02 03 01 01} \\ \texttt{01 02 03 01} \\ \texttt{01 01 02 03} \\ \texttt{03 01 01 02} \end{pmatrix}.$$

This matrix is derived from the parity-check matrix of an MDS or *maximally distance separable* code. A discussion of MDS codes would take us too far off-topic so more information can be found in [185, 189, 715]. The important feature of the matrix used in the AES is that we can associate what is termed a *branch number* β with the matrix. The branch number β describes the following property. Suppose we have eight bytes x_0, x_1, x_2, x_3, y_0, y_1, y_2, and y_3 satisfying

$$\begin{pmatrix} \texttt{02 03 01 01} \\ \texttt{01 02 03 01} \\ \texttt{01 01 02 03} \\ \texttt{03 01 01 02} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}.$$

Then either all x_i and all y_i are zero $(0 \leq i \leq 3)$ or there are at least β nonzero values among the x_i and y_i $(0 \leq i \leq 3)$.

It might not be immediately clear why this is important. However, we can at least motivate the use of an MDS matrix in terms of the *avalanche of change*. Suppose we have two inputs to the matrix M that are given by

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \text{ and } \begin{pmatrix} x'_0 \\ x'_1 \\ x'_2 \\ x'_3 \end{pmatrix}.$$

Further, imagine that these two inputs are so similar that $x_1 = x'_1$, $x_2 = x'_2$, and $x_3 = x'_3$. We then have that

$$\begin{pmatrix} \texttt{02 03 01 01} \\ \texttt{01 02 03 01} \\ \texttt{01 01 02 03} \\ \texttt{03 01 01 02} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} \text{ and}$$

$$\begin{pmatrix} \texttt{02 03 01 01} \\ \texttt{01 02 03 01} \\ \texttt{01 01 02 03} \\ \texttt{03 01 01 02} \end{pmatrix} \begin{pmatrix} x'_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y'_0 \\ y'_1 \\ y'_2 \\ y'_3 \end{pmatrix}.$$

Since multiplication by M is a linear operation over GF(2) we can write

$$\begin{pmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{pmatrix} \begin{pmatrix} x_0 + x'_0 \\ x_1 + x_1 \\ x_2 + x_2 \\ x_3 + x_3 \end{pmatrix} = \begin{pmatrix} y_0 + y'_0 \\ y_1 + y'_1 \\ y_2 + y'_2 \\ y_3 + y'_3 \end{pmatrix} \text{ and so}$$

$$\begin{pmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{pmatrix} \begin{pmatrix} x_0 + x'_0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} y_0 + y'_0 \\ y_1 + y'_1 \\ y_2 + y'_2 \\ y_3 + y'_3 \end{pmatrix}.$$

Since the branch number is 5, this means that all the values $y_0 + y'_0$, $y_1 + y'_1$, $y_2 + y'_2$, and $y_3 + y'_3$ must be nonzero and that the two outputs generated from two very closely related inputs must differ in every byte.

Of course, we could have deduced this simple case by observing that there are no zero entries in the first column of the matrix M. However, even in the slightly more complicated case of two nonzero input bytes we can be sure that at least three of the output bytes must be nonzero. Thus, when two inputs to the `MixColumns` layer are very similar in terms of the bitwise difference between them, the outputs will be very different and this should help to amplify even a small amount of "change" or "difference" over successive rounds of the cipher.

Together the `ShiftRows` and `MixColumns` operations help to ensure that the number of what are termed *active* S-boxes is either large to begin with, or becomes large very soon. This forms the basis for the very sound security offered by the AES against differential and linear cryptanalysis.

3.1.5.3 AES Performance

It is worth considering the performance of the AES. We have seen that it is heavily structured and all operations are essentially byte operations. Even the bitwise operations within the S-box are hidden from view since the entire S-box substitution can be accomplished with a 256-byte lookup table. This will make the cipher suitable for old, eight-bit processors, and the opportunities for parallelism will mean that performance on more modern 32- and 64-bit processors needn't be hampered. Note that even the coefficients chosen in the MDS matrix M are very simple and suited for implementation in limited environments, though we should note that the coefficients are not quite so nice for M^{-1} in the decryption operation.[4] If there is the luxury of a reasonable amount of memory (as is typical these days) then there are tricks that allow much of the encryption (and decryption) process to be performed by table lookups. All in all, it is a very nice cipher for the implementor and it has a very smooth performance profile across a wide range of platforms.

[4] Junod and Vaudenay [341] suggest that different choices of MDS matrix might be better in this respect.