COMP4121 Project

AES Encryption and Cryptanalysis

1 Background

1.1 Motivations – The Data Encryption Standard

The Data Encryption Standard (DES) is a symmetric-key block cipher. It was developed by researchers at IBM, and submitted to the National Institute of Standards and Technology (NIST) (formerly the National Bureau of Standards) in the US following a request for a standard to use for the protection of unclassified government data. It was officially endorsed in 1977, being published as a standard in the Federal Information Processing Standards (FIPS) – FIPS 46. A total of three revised standards (FIPS 46-1 through 46-3) were published before the standard was withdrawn in 2005 [28], superseded by the Advanced Encryption Standard (AES).

DES was quickly put under the scrutiny of cryptography researchers, due to the refusal of NIST to disclose the results of their own analysis of the security of the algorithm. Initial analysis of DES revealed [18] underlying structures in the specific components used to scramble bits, which were rumoured to have been inserted as back-doors by the NSA. The selection criteria for the various permutation and substitutions of the algorithm were classified by IBM on request of the NSA, with Tuchman, the team lead for DES project at IBM, stating [21] that

> We developed the DES algorithm entirely within IBM using IBMers. The NSA did not dictate a single wire!
>
> - Walter Tuchman

However, documents declassified by the NSA in 2013, in response to a freedom of information act request, referenced their involvement in the design process, stating [20]

> NSA worked closely with IBM to strengthen the algorithm against all except brute-force attacks and to strengthen substitution tables, called S-boxes. Conversely, NSA tried to convince IBM to reduce the length of the key from 64 to 48 bits. Ultimately they compromised on a 56-bit key.
>
> - Thomas Johnson

In an email allegedly by Whitfield Diffie [13], the bias of the author was brought into question, stating that Charles Bigelow mentioned 48-bits being sufficient, but not necessarily to convince IBM.

1.1.1 Algorithm Overview

The DES algorithm begins by taking a 64-bit block, the plaintext to encipher, and a 64-bit key. The key itself is stripped of 8 bits (the last bit of each byte), which are odd parity bits for error checking, giving an effective key length of 56 bits. The algorithm has two main components: the key schedule; and the actual cipher mechanism. Overall, the algorithm involves an initial permutation of the plaintext, 16 rounds of manipulation, and a final permutation (which is the inverse permutation of the initial permutation). Each round uses a new subkey, derived from the key through the key scheduler, in order to manipulate the text in differing ways each round (see figure 1). The process for decryption is the same, but with the order of the subkeys flipped (so the last subkey is used first, etc.).

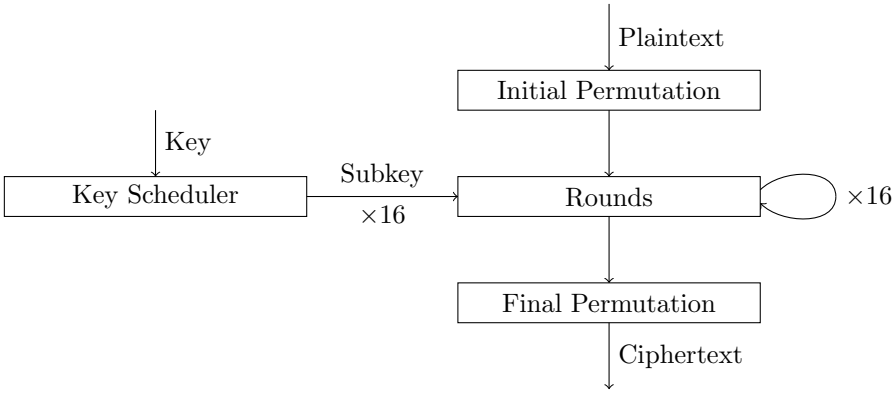

Figure 1: Basic overview of DES

The purpose of the initial and final permutations actually has nothing to do with the cryptographic security of DES [32], rather it is a consequence of how the DES algorithm was implemented in hardware. When the standard was introduced, hardware implementations loaded the 64-bit plaintext big-endian, and one byte at a time simultaneously through 8 pins. The way the chip stored the plaintext, and similarly outputted the ciphertext, was included as part of the standard, but many software implementations of DES ignored the initial and final permutations, as it served no purpose at the software level, though such implementations

couldn't be considered DES.

The key scheduler takes the now 56-bit key and produces the 16 subkeys used in each round. To do this, it first permutes the key using a specified permutation. The purpose of this is diffusion - a concept in cryptography which involves making each bit of input affect as many output bits as possible. Simple permutations don't do this directly, but the effect of permuting the key cascades through the subkey generation process. The key is then broken into two halves, both of 28-bit length, which are handled independently. At each round, the subkey is generated by first cycling each half either one or two bits to the left. For example, given the two halves $101\cdots10$ and $110\cdots01$, each half is cycled one bit to the left independently, to give $01\cdots101$ and $10\cdots011$ respectively. Some rounds shift one bit, whereas other shift two, so that overall the 28-bit halves are cycled through completely, with a total of 28 left shifts over the 16 rounds. After each shift, the subkey for the corresponding round is generated by taking each half and permuting them together. This permutation is the same for each round. The resulting subkey is 48 bits long, with 4 bits from each half not being used in the subkey. The shifts are chosen such that each bit of the key is used in an average of 14 subkeys. See figure 2 for an overview of this process.

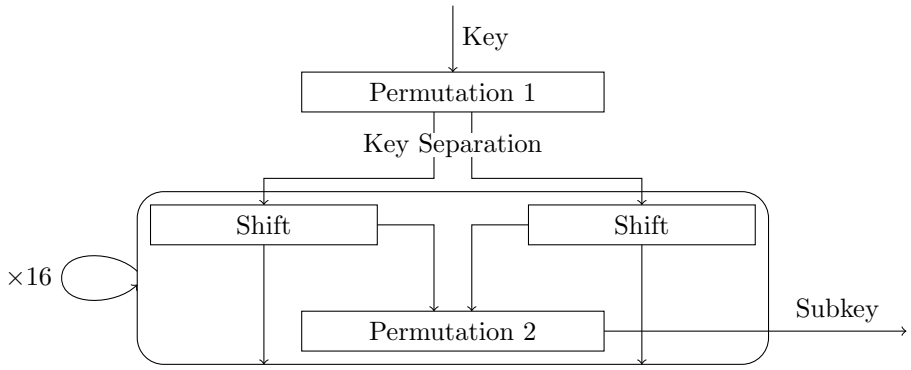

Figure 2: Overview of the key scheduler

DES is a Feistel cipher, which is a type of cipher that alternates acting on two halves of the text at each round, and mixes them at each step. See figure 3 for a depiction of the structure. At each round, the second half of the text is passed through some transformation, referred to as a Feistel function, along with the subkey for that round. The output of this function is then XOR'd with the first half of the text, and becomes the second half of the next round. The original second half of the message is preserved, and becomes the first half for the next round. After the final round, the halves aren't swapped, and give the resulting ciphertext. For a basic Feistel cipher, this is the end of the process, but for DES the result is passed through the final permutation before the ciphertext is obtained.

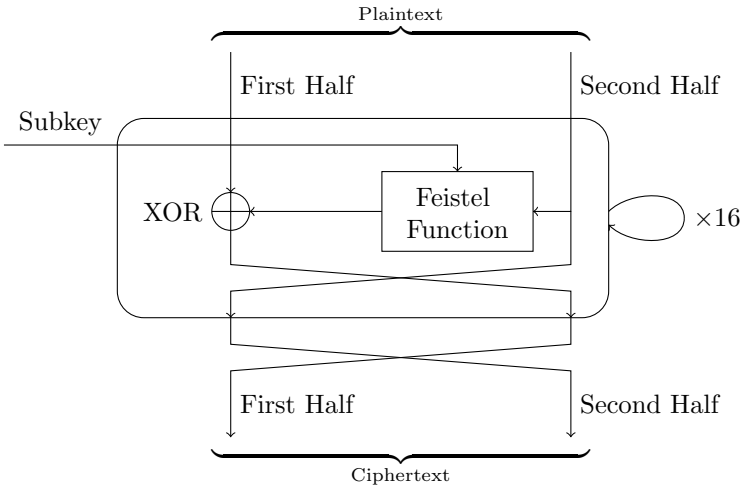

Figure 3: The basic structure of a Feistel cipher

The Feistel function used in DES is where much of the confusion – a concept in cryptography which involves obscuring the relationship between input and output – occurs. The 32-bit message (one half of the input of the round) is expanded to 48 bits by duplicating half the bits. Then, it is XOR'd with the 48-bit subkey for the round, and the result is passed through a set of substitutions, called "S-boxes". Each of these S-boxes acts on six of the bits, with a total of eight S-boxes to substitute the whole expression. The output of each S-box is four bits, so the entire substitution takes the 48-bit expression and gives a 32-bit output. Finally, this 32-bit expression is passed through a permutation. See figure 4.

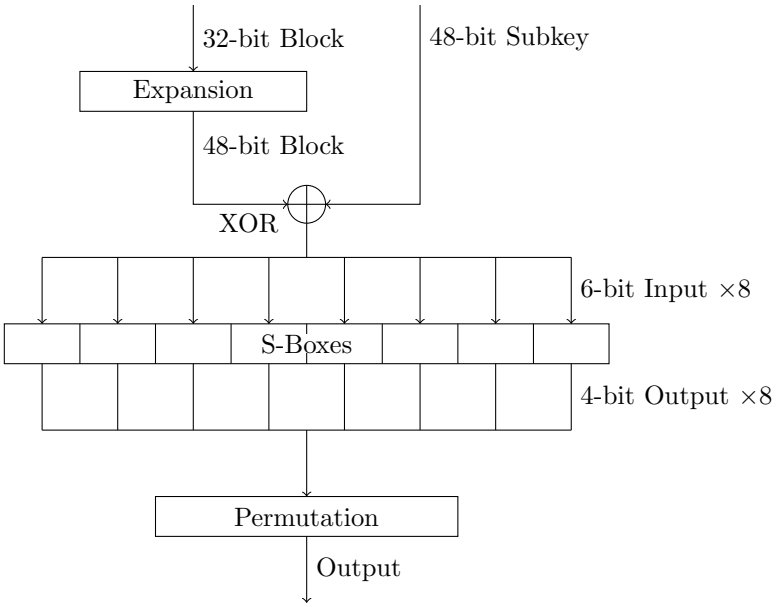

Figure 4: The DES Feistel function

1.1.2 Cryptanalysis

The DES algorithm has a complementary property which reduces the number of keys required for a bruteforce search. Let the Feistel function of DES be f, and its expansion function, S-Boxes, and permutation be E, S, and P respectively. Write L_i and R_i for the inputs of the i^{th} round, and k_i for the subkey, generated from k. Then the result of round i is the tuple

$$(R_i, L_i \oplus f(R_i, k_i)) .$$

We can write

$$f(R_i, k_i) = P(S(E(R_i) \oplus k_i)).$$

Now let $\overline{x}$ denote the (one's) complement of x. We have

$$\begin{aligned}
\overline{L_i} \oplus f(\overline{R_i}, \overline{k_i}) &= \overline{L_i} \oplus P(S(E(\overline{R_i}) \oplus \overline{k_i})) \\
&= \overline{L_i} \oplus P(S(\overline{E(R_i)} \oplus \overline{k_i})) \\
&= \overline{L_i} \oplus P(S(E(R_i) \oplus k_i)) \\
&= \overline{L_i \oplus f(R_i, k_i)},
\end{aligned}$$

where we have used the facts $\overline{x} \oplus \overline{y} = x \oplus y$, $\overline{x} \oplus y = \overline{x \oplus y}$, and that since E only duplicates the bits of R_i, $E(\overline{R_i}) = \overline{E(R_i)}$. Since the key scheduler only uses shifts and permutations to generate each subkey, we know that the subkeys of $\overline{k}$ are $\overline{k_i}$. Finally, the initial and final permutations don't affect the complementation. An inductive argument then gives us that if we encrypt $\overline{x}$ with key $\overline{k}$ and get say y as output, and similarly get y' as output using message x and key k, then $\overline{y} = y'$. That is, the complement key encrypts (and thus decrypts, since we have a symmetric algorithm) the complement message. This immediately decreases the search space of keys for DES from 2^{56} to 2^{55} for an exhaustive search. At the time, this was sufficient, requiring about 36 quadrillion keys to be checked. However, as technology got faster, this became feasible to break, and in 1997 the DESCHALL Project, in response to a challenge by RSA Security with a prize of $10,000, used distributed computing to break a DES encrypted message using a simple bruteforce approach [9]. This confirmed concerns about how long DES would hold up to growing technological advancements made by Whitfield Diffie and Martin Hellman in 1977 [14].

The first known successful (theoretical) attack on full 16-round DES was published in 1993 by Biham and Shamir [3]. The attack involved differential cryptanalysis, a statistical technique which analyses the differences in output for many inputs. Biham and Shamir's attack required approximately 2^{36} ciphertexts

to be analysed taken from a pool of 2^{47}, a big improvement over the 2^{55} required for bruteforce attacks, and was designed to be highly parallelisable. A big advantage to this attack was that it allowed key changes during its execution.

> An interesting feature of the new attack is that it can be applied with the same complexity and success probability even if the key is frequently changed and thus the collected ciphertexts are derived from many different keys. The attack can be carried out incrementally, and one of the keys can be computed in real time while it is still valid. This is particularly important in attacks on bank authentication schemes, in which the opponent needs only one opportunity to forge a multi-million dollar wire transfer, but has to act quickly before the next key changeover invalidates his message.
>
> - Biham and Shamir

Another method, discovered by Matsui in 1992, and implemented against DES in 1994 [24], involved linear cryptanalysis. This method involves attempting to find linear approximations to the cipher's results, and requires 2^{43} (in contrast to 2^{47}) ciphertexts to analyse. These attacks still remained infeasible, and eventually DES was cracked by a simple bruteforce, emphasising the need for longer encryption keys.

In 1998, the Electronic Frontier Foundation succeeded in cracking a DES encrypted message in 56 hours using a machine funded for $210,000 [15], and soon after, in collaboration with distributed.net, cracked another in under a day. At this point, it was possible for any private entity with a sufficient budget to break DES encryption.

1.2 The Selection Process

In 1992, NIST proposed reaffirmation of DES [26], accepting comments from the public regarding its suitability as a standard. In 1993, it was reaffirmed as FIPS 46-2. Five years later, it was revised as FIPS 46-3, and in 2005, the standard was withdrawn. By this point, AES had already been standardised, and DES had become obsolete.

In September 1997, recognising the need for a more secure standard, NIST put out a call for candidate algorithms as the new Advanced Encryption Standard [27]. The minimum submission requirements for a candidate algorithm were:

1. The algorithm uses a symmetric key – the same key is to be used for encryption and decryption;
2. The algorithm is a block cipher – the message to be encrypted is broken into blocks of specific sizes, and each block is encrypted separately;

3. The algorithm must support at least keys of length 128, 192, and 256 bits, with block size of 128 bits.

The candidates were to be evaluated based on security, computational cost, flexibility, and simplicity. The deadline for submission was June 15, 1998, and comments on the algorithms submitted were open until April 15, 1999. In October of 1999, NIST released a report [29] with a summary of the comments and analysis, and revealing the five algorithms to progress to the second round: MARS, RC6, Rijndael, Serpent, and Twofish. At the same time, comments for the second round were opened to the public, closing May 15, 2000. On October 2 of the same year, NIST announced [30] Rijndael as the algorithm chosen to be the Advanced Encryption Standard. In 2001, NIST submitted a draft standard describing AES, which was then published as FIPS 197 [31] after public comment.

Unlike DES, AES was selected in a public selection process, and Rijndael was developed by researchers unaffiliated with the NSA. This has contributed to the recognition of AES as a secure algorithm by the public cryptography community.

2 The AES Algorithm

The basic structure of the AES algorithm is similar to DES (see figure 5). There are 10, 12, or 14 rounds, depending on the length of the key (128, 192, and 256 bits respectively), each taking a subkey derived from the key. The main difference is how each round is completed, since AES is not a Feistel cipher. Rather, each round applies substitutions and permutations to the whole block of text directly, then mixes in the round's subkey, though the last round is slightly different, in that one of these permutations is removed.

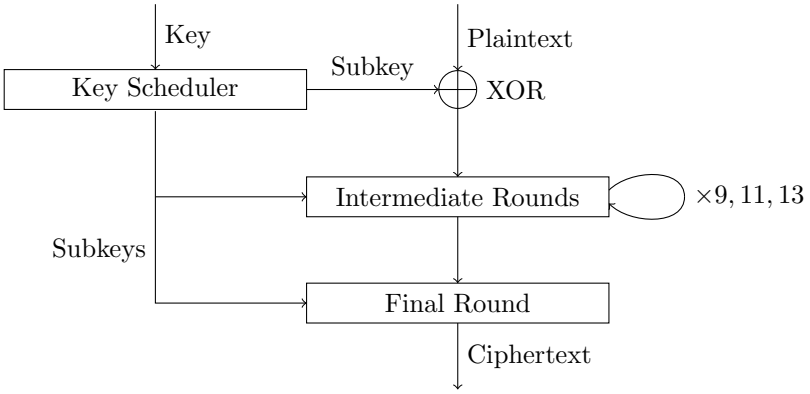

Figure 5: An overview of AES

The mathematical foundations of AES come from the theory of finite fields; specifically, the field $GF(2^8) \simeq GF(2)[x]/\langle x^8 + x^4 + x^3 + x + 1\rangle$. That is, the field consisting of all binary polynomials, modulo $p(x) = x^8 + x^4 + x^3 + x + 1$. Most operations can be expressed from the perspective of bit manipulation. As an

example of arithmetic in this field, let's calculate $2 \cdot 11010111_2 + 00110011_2$, where a_2 signifies that a is in binary. We write 11010111_2 as a polynomial

$$x^7 + x^6 + x^4 + x^2 + x + 1.$$

Now, $2 = 10_2$ is x as a polynomial, so $2 \cdot 11010111_2$ is

$$x(x^7 + x^6 + x^4 + x^2 + x + 1) = x^8 + x^7 + x^5 + x^3 + x^2 + x.$$

This is beyond our 8-bit limit, so we reduce it modulo p, by adding it to the expression. This works since $x^n + x^n = 0$ in our field (the coefficients are taken modulo 2). So, modulo p, we have

$$\begin{aligned} x^8 + x^7 + x^5 + x^3 + x^2 + x &= x^8 + x^7 + x^5 + x^3 + x^2 + x + x^8 + x^4 + x^3 + x + 1 \\ &= x^7 + x^5 + x^4 + x^2 + 1. \end{aligned}$$

Finally, we can add 00110011_2, which is $x^5 + x^4 + x + 1$, giving us

$$x^7 + x^5 + x^4 + x^2 + 1 + x^5 + x^4 + x + 1 = x^7 + x^2 + x.$$

That is, $2 \cdot 11010111_2 + 00110011_2 = 10000110_2$. All arithmetic is like this in AES.

2.1 The Key Scheduler

The key scheduler for AES is the most complicated part of the algorithm. The key scheduler runs in rounds, similar to the main algorithm. Each round produces a new derived key, which is the same length as the original key. For 128-bit AES, these are exactly the subkeys used in the algorithm, but for 192- and 256-bit AES, the subkeys are generated by taking chunks out of the derived keys. For example, in 192-bit AES, the first derived key is 192-bits long. The first 128 bits of this are the first subkey. Then, the last 64 bits are then concatenated with the first 64 bits of the second derived key to get the next subkey, and so on. See figure 6 for an example of this process. The key scheduler of 128-, 192-, and 256-bit AES have 11, 9, and 8 rounds respectively, as although the number of subkeys required increases, the length of the derived keys does as well.

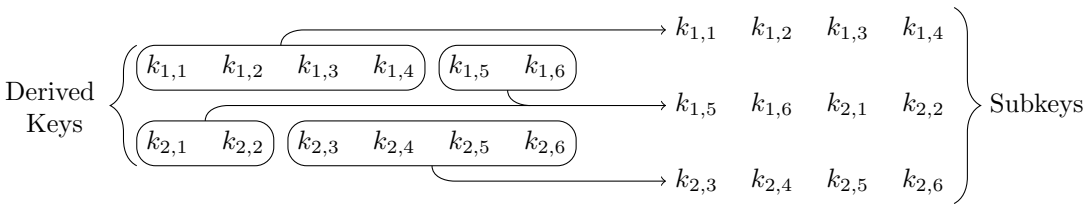

Figure 6: Example of turning derived keys into subkeys; each $k_{i,j}$ is a 32-bit block

We will write $k_{i,j}$ for the j^{th} block of the i^{th} derived key. The last block of each round is used in the subsequent round, but is first cycled left by one byte, passed through a byte-wise substitution, and finally XOR'd with a constant determined by the current round number. We will write f_i as the function which does this process, using the constant corresponding to round i. Let n be the number of 32-bit blocks in the key. We begin by letting $k_{1,j}$ be the j^{th} 32-bit block in the original key. Then, recursively define

$$k_{i,j} = \begin{cases} k_{i-1,1} \oplus f_i(k_{i-1,n}), & j = 1; \\ k_{i-1,j} \oplus k_{i,j-1}, & j \neq 1. \end{cases}$$

For every block after the first, this is just an XOR between the corresponding block in the previous round, and the previous block in the current round. The first block is an XOR between the corresponding block in the previous round and the result of passing the last block of the previous round through f_i. This process is shown in figure 7.

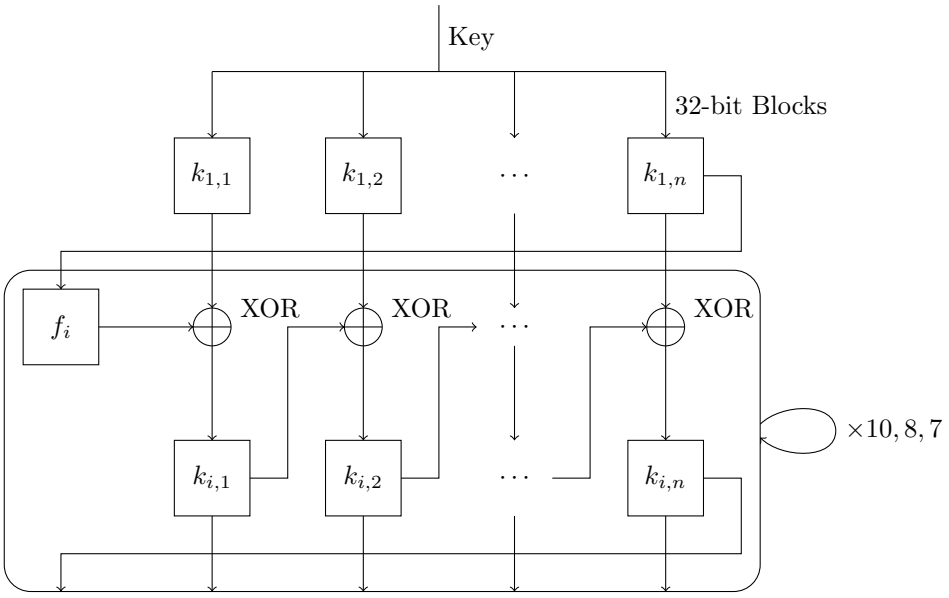

Figure 7: The key schedule of AES; subkeys are extracted from the $k_{i,j}$

The constant used for each round, is derived as follows. Start with $c_1 = 01_{16}$, writing a_{16} to denote that a is expressed in hexadecimal. Then each successive constant is $c_i = 2c_{i-1}$ if $c_{i-1} < 80_{16}$, or $c_i = (2c_{i-1}) \oplus 1\mathrm{B}_{16}$ otherwise, which ensures we don't overflow. The constant used in round i is then c_i concatenated with 000000_{16}. That is, c_i followed by three zero bytes.

The substitution used is derived from the multiplicative inverses of the bytes within the field $GF(2^8)$. Inversion has been studied extensively in mathematical literature, so alone it can't be considered sufficiently "complex" enough to be all the confusion the key scheduler adds, however, it is highly non-linear (see figure 8). The substitution involves taking the multiplicative inverse (taking the "inverse" of 0 to be 0), and passing it through an affine transformation, which ensures the transformation in robust against standard algebraic techniques. The affine transformation can be written, denoting a left cyclic shift with $\lll$, as $b \oplus (b \lll 1) \oplus (b \lll 2) \oplus (b \lll 3) \oplus (b \lll 4) \oplus 63_{16}$, where b is the byte.

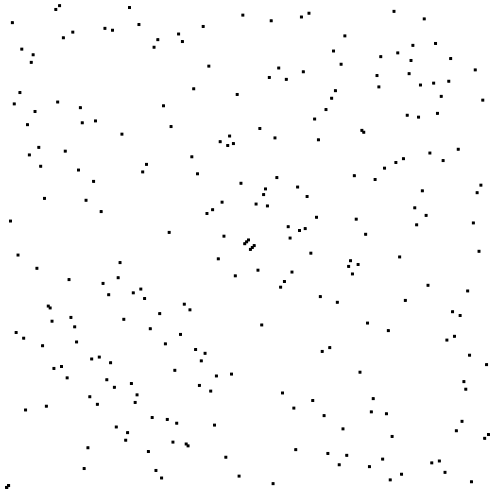

Figure 8: A scatter plot of the multiplicative inverses of elements of $GF(2^8)$

2.2 The Rounds

Each round applies the same transformations, except the last, which only differs in the removal of one transformation. The 128-bit block is acted upon as a 4×4 "matrix" of bytes, read column-major. That is, given a block with bytes $b_1, b_2, \cdots, b_{16}$, we represent it as the matrix

$$\begin{pmatrix} b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \\ b_4 & b_8 & b_{12} & b_{16} \end{pmatrix}.$$

The first transformation is a substitution of each byte (corresponding to an entry in the matrix), using the same substitution as the key scheduler. Then, the rows are shifted left cyclically by 0, 1, 2, and 3 bytes respectively as below:

$$\begin{pmatrix} b_1 & b_5 & b_9 & \boxed{b_{13}} \\ b_2 & b_6 & b_{10} & \boxed{b_{14}} \\ b_3 & b_7 & b_{11} & \boxed{b_{15}} \\ b_4 & b_8 & b_{12} & \boxed{b_{16}} \end{pmatrix} \mapsto \begin{pmatrix} b_1 & b_5 & b_9 & \boxed{b_{13}} \\ b_6 & b_{10} & \boxed{b_{14}} & b_2 \\ b_{11} & \boxed{b_{15}} & b_3 & b_7 \\ \boxed{b_{16}} & b_4 & b_8 & b_{12} \end{pmatrix}.$$

After this, the columns are mixed using a linear transform, taking

$$\begin{pmatrix} b_1 & b_5 & b_9 & b_{13} \\ b_6 & b_{10} & b_{14} & b_2 \\ b_{11} & b_{15} & b_3 & b_7 \\ b_{16} & b_4 & b_8 & b_{12} \end{pmatrix} \mapsto \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \begin{pmatrix} b_1 & b_5 & b_9 & b_{13} \\ b_6 & b_{10} & b_{14} & b_2 \\ b_{11} & b_{15} & b_3 & b_7 \\ b_{16} & b_4 & b_8 & b_{12} \end{pmatrix}$$

Here, addition is XOR, and multiplication is from the perspective of polynomials in the field. So, $3 = 11_2$ is actually $x+1$, and after multiplication, we reduce it modulo the polynomial defining the field. Each column of the resulting matrix only depends on entries from the corresponding column from before the transformation, hence the name. In the final round, this step is skipped. This operation, as well as the row-wise shifting previously, contribute diffusion to the cipher, whereas the substitution step adds confusion.

Finally, the subkey is added to the text by a simple XOR. Since the subkeys are extracted from the derived keys so that they are 128 bits long, no expansion is necessary. Here, we XOR each byte column-wise, just as we did to transform the 128-bit block into this matrix representation.

Decryption involves inverting every step. All of the transformations are invertible, so doing the entire algorithm in reverse, making sure the subkey for each step is the correct one (i.e. reverse order) will take a ciphertext and produce the plaintext. This relies on the fact that $(x \oplus y) \oplus y = x$. That is, we can apply the key twice to recover the original text, and pass that back up to the next step.

3 Cryptanalysis

There are two main types of attacks on cryptographic systems – direct attacks, and side-channel attacks. A direct attack generally involves exploiting properties of the algorithm to determine keys or messages given only varying degrees of information about the input and output of a system, treating the algorithms itself as

something that can't be interfered with directly, such as differential cryptanalysis. Generally, these kinds of attacks are of high significance if found, and are the main concern of cryptographers. Side-channel attacks are attacks aimed at systems themselves, rather than the algorithm they run. They generally capitalise on deficiencies in the physical implementations of a system, or in characteristics of the algorithms which can somehow be observed externally. An example of this is power analysis, which involves distinguishing various operations of an algorithm by their power consumption.

3.1 Direct Attacks

Theoretical attacks against AES as an algorithm have shown AES to be very secure. AES was designed specifically to hinder differential and linear cryptanalysis attacks [12], which were the main theoretical attacks against DES. The current best attack reduces AES-128 from 2^{128} time to $2^{126.2}$ time [5] using a biclique attack, which is practically no improvement. If a computer was capable of encrypting/decrypting 1,000,000,000 messages per second, it would take approximately 3×10^{21} years to find the key.
A related-key attack on AES was successfully developed to reduce AES-256 from 2^{256} bruteforce, to $2^{99.5}$ time [4]. A related key attack requires the attacker to be allowed to observe the result of encryption or decryption of multiple keys, related in some way. The relations of keys can be used to deduce the key bits, if the algorithm is simple, such as AES. Related-key attacks have been successful against RC4 [23], as many implementations simply concatenated a long key with a much shorter nonce. However, AES is used in a context where this should not occur. There is no requirement to use a different key every time, and if the key changes, there is no reason to have them related in a direct manner. As such, a related-key attack on AES is somewhat impractical, and isn't considered a security breach.

3.1.1 Differential Cryptanalysis

Differential cryptanalysis, independently discovered by the NSA, IBM, and Biham and Shamir, is an attack which uses pairs of plaintexts and their ciphertexts to deduce the key used in an encryption algorithm. If we write x and x' for two inputs (either plaintexts, or somewhere else in the algorithm), and k for a key, we have that

1. If f is linear, then $f(x \oplus x') = f(x) \oplus f(x')$,
2. $(x \oplus k) \oplus (x' \oplus k) = x \oplus x'$.

If we instead consider differences (i.e. XOR between them) of plaintexts, every step except the substitution step keeps our difference. These substitutions are what we analyse to apply differential cryptanalysis [19].

To do this, we can find a distribution of the outputs of the substitution depending on our input difference. Since the substitution isn't linear, we can't determine exactly what the result of the substitution is, but we can try every possible pair, and find the input and output differences after passing them through the substitution. Formally, if S is our substitution, we calculate

$$s_{i,j} = \left|\{(x, x') : x \oplus x' = i, S(x) \oplus S(x') = j\}\right|,$$

and form a table from these. As a simple example, consider a substitution acting on three bits, defined by

	x00	x01	x10	x11
0xx	001	011	110	111
1xx	010	100	101	000

That is, 001 is mapped to 011, etc. Now, we consider every possible input pair (of which there are 64), and calculate the input and output XORs. Counting these gives us

	Output XOR							
Input XOR	000	001	010	011	100	101	110	111
000	8	0	0	0	0	0	0	0
001	0	2	2	0	0	2	2	0
010	0	0	0	0	4	0	0	4
011	0	2	2	0	0	2	2	0
100	0	0	0	4	0	0	0	4
101	0	2	2	0	0	2	2	0
110	0	0	0	4	4	0	0	0
111	0	2	2	0	0	2	2	0

Now, this can tell us the probability of a certain output from a substitution, given an input. For example, If the initial plaintext has difference 010, then the output from the substitution is either 100 or 111, each with probability 0.5 assuming uniform input distribution. Since we are dealing with differences, and $(x \oplus k) \oplus (x' \oplus k) = x \oplus x'$, these are all independent of the key. We will write $x \xrightarrow{p} y$ to denote that there is a probability of p for an input difference x to become y after substitution.

A n-round characteristic is a collection of an input, output, and intermediate values, where the output is after n rounds. The probability of the characteristic is the probability of each step in the algorithm producing the corresponding values. This is under an independence assumption, which gives us some error, but is negligible [11]. For example, consider a simplified cipher which has only three substitution steps, each with an XOR

with a key beforehand. Then a sample 2-round characteristic is

$$001 \overset{0.25}{\rightarrow} 010 \overset{0.5}{\rightarrow} 100.$$

The probability of this characteristic is

$$0.25 \times 0.5 = \frac{1}{8}.$$

In general, we try to find the best possible characteristic (with respect to probability) which covers all but one of the rounds of the cipher for varying input/output differences.

Now, we can try to attack the key. We can find many pairs of plaintext with the same difference as our sufficiently likely $(n-1)$-round characteristic. Encrypt them, and get their output difference. Since any pair that satisfies our characteristic must only affect a few of the S-boxes, any that have a non-zero difference in the bits from S-boxes not affected by our characteristic can be discarded. For each of the possible bits of the key corresponding to the affected S-boxes, we decrypt the bits from the ciphertexts to get the output of the $(n-1)^{\text{th}}$ round, and take the difference. If this difference matches the value of the characteristic, then we count it towards the partial key we used to decrypt the last round, and otherwise count it against the partial key. After running this many times, we will have a frequency distribution of which partial keys decrypted gave us the right value, and can take the largest frequency as our guess for those bits of the key.

In our case, there is only one S-box, so to determine the key we would end up bruteforcing, but for algorithms with multiple S-boxes, the number of bits required to bruteforce can be much less [19] (dependent on the cipher structure).

When Rijndael was proposed as the new AES, differential cryptanalysis was widely known, and its operations were specifically designed to hinder it. To do this, every round is designed so that the number of changes in bits is at least five [10], which propagates along all 10 or more rounds of the algorithm. This results in no 4-round characteristic having probability greater than 2^{-150}, which forces differential attacks on the algorithm to require more plaintext/ciphertext pairs to be analysed than bruteforce would require.

There have been successful attacks on reduced-round versions of AES using differential cryptanalysis. A recently published attack on 5-round AES reduces the number of ciphertexts required to 2^{32} [1]. The attack relies on the ability to choose texts to encrypt/decrypt, but requires no knowledge on the S-box used, so can be used against modified implementations of AES with different diffusion properties. Although these attacks can provide insight into possible attack vectors of a cipher, the techniques don't easily generalise, and often are completely inapplicable to the full-round versions of AES.

3.1.2 Linear Cryptanalysis

Linear cryptanalysis, discovered by Matsui [24] against DES, has a similar design to differential cryptanalysis. Rather than focusing on how differences in input propagate through the algorithm, linear cryptanalysis attempts to find linear approximations to the substitution boxes. Just as with differential cryptanalysis, we can discount most linear steps, only worrying about the effect of the substitution, and how the changes in plaintext propagate through the system. The technique relies on the non-random nature of the substitution boxes and a lemma by Matsui, called the "Piling-Up Lemma". It states that given n many independent binary random variables $X_1, \cdots, X_n$ with probabilities

$$\mathbb{P}(X_i = 0) = \frac{1}{2} + \varepsilon_i,$$

we have

$$\mathbb{P}(X_1 \oplus \cdots \oplus X_n = 0) = \frac{1}{2} + 2^{n-1} \prod_{i=1}^{n} \varepsilon_i.$$

These ε_i values are called biases, and represent how much each variable differs from a uniform distribution, and in which direction (i.e. more likely to be 0, or 1).

Now, instead of finding a characteristic for $n-1$ rounds, we find an approximation for the relationship between some inputs and the value of the state at the end of round $n-1$. If we have such an approximation, we can apply a statistical analysis similar to that of differential cryptanalysis to determine partial subkeys, and work backwards to determine the entire key.

We will use the same substitution as used to exemplify differential cryptanalysis, but instead of finding the number of input/output XOR pairs, we find relationships between the different (XOR) sums of input bits and output bits. Further, we de-bias these counts by subtracting half the number possible values.

First, some notation. Our input bits into the substitution will be represented as X_1, X_2, X_3, modelling three binary random variables. Similarly, our output bits will be Y_1, Y_2, Y_3. Let X_{abc} be the expression $aX_1 \oplus bX_2 \oplus cX_3$, so that X_{101} represents $X_1 \oplus X_3$, and similarly for Y_{abc}. For every possible input X_1, X_2, X_3 with output Y_1, Y_2, Y_3, we count the number of times $X_{abc} = Y_{a'b'c'}$ for each a, b, c, a', b', c'. Each entry in the table below has 4 subtracted to de-bias the values.

	Y_{000}	Y_{001}	Y_{010}	Y_{011}	Y_{100}	Y_{101}	Y_{110}	Y_{111}
X_{000}	+4	0	0	0	0	0	0	0
X_{001}	0	0	0	-4	0	0	0	0
X_{010}	0	0	0	0	+2	-2	-2	-2
X_{011}	0	0	0	0	+2	+2	+2	-2
X_{100}	0	-2	-2	0	0	-2	+2	0
X_{101}	0	+2	+2	0	0	-2	+2	0
X_{110}	0	-2	+2	0	+2	0	0	+2
X_{111}	0	-2	+2	0	-2	0	0	-2

The -4 for (X_{001}, Y_{011}) represents that $4 - 4 = 0$ of the possible input/output pairs have $X_3 = Y_2 \oplus Y_3$, and that we have a bias of $\frac{-4}{8} = -\frac{1}{2}$ (i.e. its probability is zero). We care mostly about those with large magnitude, since they indicate that there is a strong correlation between the input/output sums differing from the average distribution.

Now to construct an approximation, we can find something akin to a characteristics, but by composing relations of the form $X_{i_1} \oplus \cdots \oplus X_{i_k} = Y_{j_1} \oplus \cdots Y_{j_m}$. If we denote the input and output of round i by X^i and Y^i respectively, we can form a system

$$\begin{aligned} X_2^1 &= Y_1^1 && \text{with probability } \frac{6}{8}, \\ X_1^2 &= Y_1^2 \oplus Y_3^2 && \text{with probability } \frac{2}{8}. \end{aligned}$$

By the piling-up lemma, this has bias

$$2\left(\frac{6}{8} - \frac{1}{2}\right)\left(\frac{2}{8} - \frac{1}{2}\right) = -\frac{1}{8}.$$

Now, we can do a similar analysis as in our differential cryptanalysis example. Our approximation links the first bit of the input with the first and last bit of the output of the second-last round. If we now try every partial key affected by these bits (again, in our simple case it's all of them, but for multiple S-boxes, the approximations can be chosen to avoid certain S-boxes, reducing the search space significantly), and decrypt ciphertexts to the second last round using each partial key, we can apply our approximation, and count the number of texts which agree or disagree with the approximation. The bias of each key can be calculated by determining how many texts agree for that partial key compared to true randomness (for which we'd expect half to), and the partial key which has largest magnitude in bias is most likely to be correct. Multiple different approximations over many samples can reveal the entire key.

Just like the case with differential cryptanalysis, AES is resistant to linear cryptanalysis, with no 4-round "characteristic" (in the linear sense) having bias higher than 2^{-75} [10], which is sufficient to ensure linear cryptanalysis attacks require more data than exhaustive search, as the amount of ciphertext required is approximately proportional to ε^{-2} [24], where ε is the bias of the approximation. Larger bias increases the amount of information we gain from that approximation, decreasing the number of texts needed to get accurate measurements.

3.2 Side-Channel Attacks

3.2.1 Simple Power Analysis (SPA)

Simple power analysis is the process of monitoring changes in power usage of a microcontroller running an algorithm, and using characteristics of its implementation to derive secrets [8]. As an algorithm runs, the amount of power required to complete an operation varies. For example, consider the following pseudocode.

```
begin
    for i := 0 to 20 do
        if i % 2 == 0 do
            long_calc(i);
        else
            short_calc(i);
        end;
    end;
end;
```

If `long_calc` takes about twice as much computational effort as `short_calc` for example, then we can see differences in the power consumption, as illustrated in figure 9. This can be leveraged when analysing hardware which implements various algorithms, to unobtrusively extract secrets, like keys. A simple example of this is using SPA to leak the exponent used in RSA decryption. In this case, the secret is d, which is a private exponent. n is the public modulus, and x is the public message to be encrypted. To decrypt the message, RSA calculates

$$y = x^d \pmod{n}.$$

A naïve implementation of this is to use the method of exponentiation by squaring, which evaluates an integer power by finding

$$\mathrm{Pow}(x,d) = \begin{cases} x, & d = 1; \\ \mathrm{Pow}\left(x^2, \frac{d}{2}\right), & d \text{ is even}; \\ x \cdot \mathrm{Pow}\left(x^2, \frac{d-1}{2}\right), & d \text{ is odd}. \end{cases}$$

This reduces the exponentiation operation from $O(d)$ to $O(\log_2 d)$. However, it also introduces an overhead

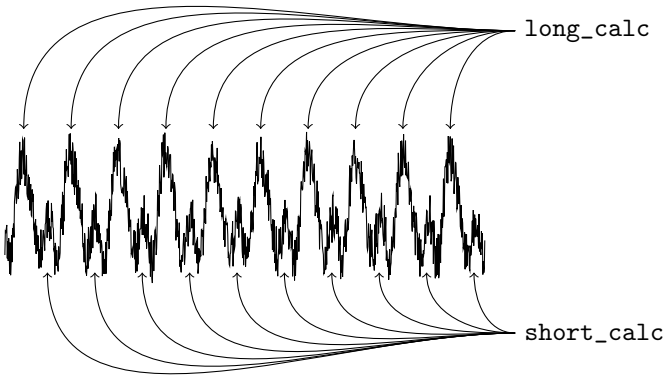

Figure 9: A simulation of power consumption over time

which can be used to leak d bit-by-bit.

As an example, suppose $d = 1101_2$. To calculate x^d, we can unroll the recursion as

$$x^{1101_2} = x \cdot (x^{110_2})^2 = \cdots = x \cdot \left(\left(x \cdot (x)^2\right)^2\right)^2.$$

Now when calculating this, we evaluate from the inside out, in the order:

1. Start with just x,
2. Square the result and multiply by x,
3. Square the result,
4. Square the result and multiply by x.

The power profile of these operations is slightly different, as shown in figure 10. By analysing the pattern of these operations, we can associate a bit to each "peak", generating the sequence 101_2, and knowing we must start with 1 originally (loading x takes negligible power), we get 1101_2, which is the private exponent, and can now be used to decrypt messages sent to the hardware.

To reduce the effect of SPA on the security of a system, random noise can be added to the system, which can obfuscate the individual operations. Further, simple conditions and control structures should be avoided [22]. Generally, the removal of these conditions, or addition of "dummy" instructions which have a similar power profile to other instructions but do nothing, can cause the performance of the hardware to decrease, as these are usually optimisations. In our RSA example, multiplying by x every time, regardless of parity, and just

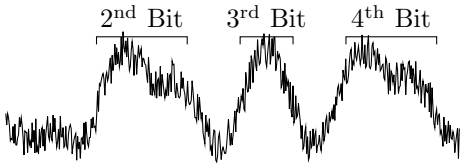

Figure 10: Simulation of exponentiation operation from the RSA example

discarding that result would mask the difference in power signature, however, it is still a relatively expensive operation, and could almost double the time the algorithm requires to decrypt a message. Noise can be circumvented by simply taking measurements over multiple decryptions and averaging them, adjusting for the slightly different power consumption of multiplications on different numbers.
SPA doesn't directly apply to AES, since the power signature doesn't involve large conditionals like RSA, but this allows us to discuss differential power analysis, which is applicable.

3.2.2 Differential Power Analysis (DPA)

In practice, SPA is not powerful enough to leak much information. If the operations cannot be clearly separated, it becomes difficult to determine anything from the power traces. Instead, differential power analysis uses the characteristics of the algorithm with measurements of the power supply, to effectively bruteforce the key using statistical correlation [8]. The main difference to regular bruteforce, is that rather than checking the entire keyspace, you can focus on parts of the key separately. To discuss DPA, we must first outline how microcontrollers store and manipulate data.
To transfer data between different parts of a microcontroller, a data bus is used. For an 8-bit microcontroller, this bus transfers 8 bits every clock cycle, by setting the corresponding lines of the bus to differing voltages, HI and LO (e.g. 5v and 0v respectively). To facilitate the changing of the voltage along these lines, electronic "switches", MOFSETs, are used, which either power, or ground a line (see figure 11). If we set up the line so that only the power line is switched, then we get the bus line swapping between LO and HI directly. When we switch the MOFSET, the power line has a spike in current, approximately linear in the difference of lines it powered between the cycles. To reduce this, the bus lines are pre-charged, so that the voltage remains somewhere in the middle of HI and LO. Just before a clock cycle occurs, a MOFSET is flipped, either powering the line to HI, or grounding it to LO, to set bus lines to the data they are transferring, and then set back to the pre-charge voltage. This decreases the amplitude of the current spikes in the power line, and introduces a smaller spike in the ground line, which provides more information about the data being transferred to someone observing these changes. Specifically, we can now observe the number of HI bits on the bus (see figure 12), rather than only the change in Hamming distance.

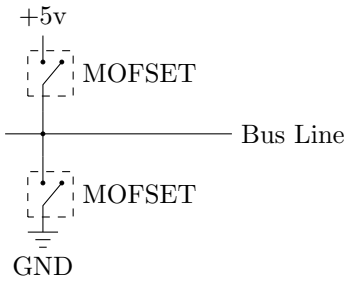

Figure 11: An example of switching a bus line on/off

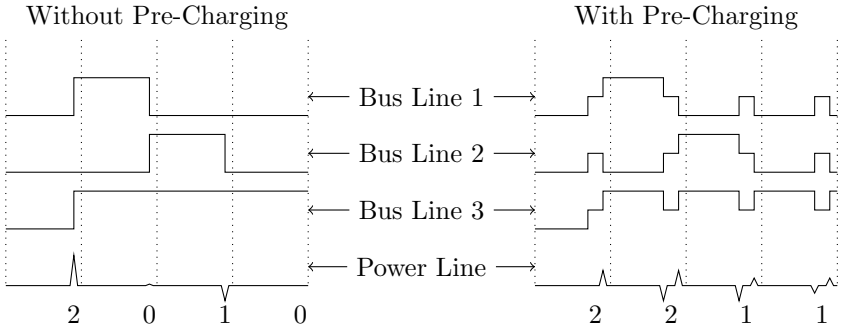

Figure 12: Difference between regular and pre-charged power traces

Now, we can use our knowledge of AES to slowly determine the key. The first 128 bits of the key are used directly in an XOR, and is then passed to a substitution table. Between these two operations, our data has to travel over a data bus, under some assumptions on the implementation model the microcontroller uses. Given enough power traces, so that they can be averaged to remove noise, there should then be some correlation between what we expect to be along the data bus, and the power consumption over time, if we have the right key byte. So, we can try random inputs to the algorithm, and collect many power traces over time, which we expect to somehow encode the key. Now, we make guesses. For every byte from 00_{16} to FF_{16}, we determine the output of an XOR with the input message using our guessed key byte. We can very easily determine the number of HI bits, and record it along with the power trace and key guess. Then, for each key guess, we can try to find a correlation between the power traces and the expected number of HI bits, at each sample point of our power trace.

Under the assumption that each key guess is correct, we can determine where in time the XOR result is most likely to have been transferred along the bus, and doing this over every possible key byte, we can determine which of the key bytes is most likely to be correct. This process can determine the first 128 bits of the key, but for AES-192 and AES-256, we then need to calculate the result of the first round, and do similar analysis. Since we know the first 128 bits of the key, we know the result of the first XOR, so there are no unknown values other than the remaining key bits. There are 256 possible guesses for a given key byte, so

if we took n samples for each byte of plaintext (so n with the same first byte, or same second byte if we're finding the second key byte, etc.), then we only need to bruteforce $256n$ combinations to get a byte. Then to recover the entire key, where r is the number of bytes in a key ($r = 16$ for AES-128), we have

$$\underbrace{256n + 256n + \cdots + 256n}_{r \text{ times}} = rn2^8 \ll 2^{2^r}.$$

In practice, these attacks can be completed within a few seconds.

Many embedded implementations of AES combine the XOR and substitution steps, since the key is embedded into the hardware, so the result of both operations doesn't change. Since the substitution is known, this doesn't affect DPA on AES, since we can adjust our expected HI bit number for each guess to suit.

Completely mitigating DPA is difficult. Rather, changes in architecture of chips can decrease the effects of the MOFSET switching on the power line, making it much more difficult to get accurate readings [16].

3.2.3 Timing Attacks

Timing attacks involve taking measurements of response times from an algorithm, either directly or remotely. For simple algorithms like RSA, these attacks can be very easy to do [6, 34]. Since RSA requires modular reduction in its calculation of the exponent, the time taken can vary significantly, depending on how many values need to be adjusted into the range $[0, n)$. To do this, many calculations of decryptions of random messages can be timed, and we can break them into different classes. Since we know n, if our messages x are chosen so that $x^2 < n \leq x^3$, and y chosen so that $y^3 < n$, then the first step will take longer for x than for y, since for x we have $x^2 < n$, which requires no reduction, but $x^3 \geq n$ does, and for y, neither operation will require reduction. Over many x and y satisfying these inequalities, we can average the timings of the two different classes. If the second (remembering the first bit must be 1 always) bit is 1, then there should be a significant difference in the average time taken for x and y decryptions, but if it's 0, then x^3 isn't calculated, so no reduction is made, and they should be similar in time. For RSA, this can be mitigated by using blinding [7], which involves taking some random r which is coprime to n, and instead decrypting xr. Since the message has how been modified by a random value, there isn't a correlation between the timings and the message, and we can reverse this after the decryption process by finding r^{-d} and multiplying. Since r is coprime to n, we know r^{-d} exists, and this calculation changes every message, so the overhead (dependent on d) can't be isolated and analysed.

For AES, these attacks are more complex [2], but the principle remains the same. Rather than attacking the time taken for a calculation, we can instead try to observe the time taken for table lookups, by using similar architecture as the victim server or chip. For efficiency reasons, as noted in section 3.2.2, AES

implementations generally combine the XOR with the subkey, and substitution table steps together. Just as with RSA, we can try varying inputs, all with some commonalities. To determine the first bit, we try many random keys, and group them together by the first byte. We can then analyse, on our own system, the timing of various keys on these inputs. As long as the architecture the algorithm is run on is similar, these should be approximately the same. If the timings for a given first byte of a key is similar to the timings from the server, then we should expect the server key to use the same byte.

There are simple mitigations, such as forcing constant-time performance for every operation. However, they generally force the algorithm to run slower, and if poorly implemented, don't stop an attacker. The timings come from how the cache handles evictions (hence why architecture is important), where parts of lookup tables are loaded into the cache, and a cache miss may evict another part of the table to load in the relevant part. As such, disabling caching can ensure any access of the lookup table is performed entirely in memory, and remains (almost) constant time, regardless of which table entry is being accessed. Alternatively, the lookup table can be loaded into the cache, and no-fill mode activated [33]. In this mode, no cache evictions are made, so there can't be any discrepancies in the timings, and there is no reduction to speed caused by misses or loading entirely from RAM. The difficult part of this approach is ensuring that no-fill mode is activated immediately after the table is loaded into cache, so that nothing is evicted, which is sometimes infeasible.

The AES-NI instruction set, implemented on almost every consumer CPU, provides an interface to encrypt and decrypt messages securely, separate from the rest of the processor. This ensures that the only data stored in its internal caches is managed by the encryption/decryption process, so no evictions can be forced, mitigating cache-timing attacks against AES. However, some implementations of this interface on modern hardware still allow for this information to be leaked, as was recently (December 2019) discovered [25], by injecting faults into the security systems isolating the encryption computations.

3.2.4 Differential Fault Analysis

Differential Fault Analysis (DFA) relies on the ability to force bits (or bytes) of an intermediate value to be faulted – modified from their true value. To apply DFA on AES, we need to be able to modify single bits [17], which is possible given an accurate laser. By polarising a laser beam circularly, a very small magnetic field is generated, which can flip bits in values stored in memory. To use this, the state of the text is faulted at the end of the second last round. Identifying where the value is stored can be difficult depending on the device, but the timing of the laser pulse can be tweaked by observing the difference in the faulted ciphertext and the expected ciphertext. If more than one byte is affected, then the fault must have occurred before columns were mixed, which doesn't happen during the last round.

Let S be the byte substitution operation, and R be the shift rows operation. Then if x_i is the i^{th} byte of the state before the last round is executed, k_i is the i^{th} byte of the final subkey, and e_i is some byte encoding the error induced by the fault on byte i, we know the resulting faulty ciphertext is given by

$$c'_j = R(S(x_i \oplus e_i)) \oplus k_j.$$

Since e has a one-bit fault, we have $c'_j = R(S(x_i)) \oplus k_j = c_j$ (the correct ciphertext) for all but one byte. This means that for exactly one byte,

$$c_j \oplus c'_j = (R(S(x_i)) \oplus k_j) \oplus (R(S(x_i \oplus e_i)) \oplus k_j) = R(S(x_i)) \oplus R(S(x_i \oplus e_i)) = R(S(x_i) \oplus S(x_i \oplus e_i)).$$

Since we know j, and how R and S move the bytes around (S doesn't), we know i, so we know where the fault occurred. We can now effectively bruteforce the possible values of x_i and e_i. We try every possible one-bit fault e_i and every possible byte x_i, and calculate $R(S(x_i) \oplus S(x_i \oplus e_i))$. If the value matches $c_j \oplus c'_j$, then we count that combination as a possible fault/byte pair. Over multiple faulty ciphertexts, we can draw up a distribution. For example, consider the following distribution (where x_i is 3 bits for simplicity).

	e_i		
x_i	001	010	100
000	1	2	0
001	2	0	2
010	0	4	1
011	0	2	2
100	4	17	6
101	2	2	2
110	2	0	5
111	0	2	0

The table clearly shows that the most likely value of x_i is 100. There will likely be small variations, as occasionally multiple bits might be flipped, giving us noise, but more trials can reduce the uncertainty.
Such a count can be determined for every byte in the second last round, until a most probable state is determined. Then

$$k_j = R(S(x_i)) \oplus R(S(x_i)) \oplus k_j = R(S(x_i)) \oplus c_j.$$

For AES-128, finding all the bytes of the last subkey is enough, but for AES-192 and AES-256 it cannot

recover the key in its entirety. Rather, this subkey can reduce the time required to bruteforce the system from 2^{192} and 2^{256} to 2^{64} and 2^{128} respectively. Realistically, AES-256 isn't affected by this technique directly, but an effective keyspace of 2^{64} for AES-192 can be bruteforced with today's technology relatively easily.

4 Conclusion

AES was designed with linear and differential cryptanalysis in mind. As such, its security against those, and derived attacks (higher-order and impossible differential, for example) is exemplary. There have been a few smaller attacks against AES proposed, such as related-key attacks, but they remain impractical. The only known "breaks" are those leveraging the specific implementations of AES in software or hardware. Many of these are hard to combat, but also require access to the hardware directly, or intimate knowledge of the implementation and host architecture to succeed. Some hardware implementations can be made somewhat resistant to attacks like these, by hardcoding the key into the logic of the chip itself.

Rather than relying on transferring the key and state bits directly, various states can be encoded into the arrangement of the logic gates themselves, and each plaintext input would then completely change how the algorithm executes, with different parts acting as states themselves. This so-called "white-box" security can be difficult in practice, and requires handcrafting chips for each key, so is impractical in many situations, but software implementations of this kind of design (using jumps to sections of code as defined by the address of a variable which is modified directly, for example) have started to become popular, as creating these kinds of software implementations can be done automatically, and there is no associated cost with switching between them if a new key is needed.

The prevalence of side-channel attacks, especially with the recent Spectre and Meltdown vulnerabilities, has highlighted how important it is for software and hardware designers to work together when considering how algorithms should be made. Often, software designers don't consider the implementation of algorithms in hardware, which allows others with more knowledge of the intricacies of hardware implementations to abuse oversights to break security and leak secrets.

References

[1] Bardeh N., R. S. Practical Attacks on Reduced-Round AES. In *Lecture Notes in Computer Science, vol. 11627* (2019), AFRICACRYPT '19, pp. 297–310.

[2] Bernstein D. Cache-Timing Attacks on AES, 2005. `http://cr.yp.to/papers.html#cachetiming`, Last accessed on December 5, 2019.

[3] Biham E., S. A. Differential Cryptanalysis of the Full 16-round DES. In *Lecture Notes in Computer Science, vol. 740* (1993), CRYPTO '92, pp. 487–496.

[4] Biryukov A., K. D. Related-key Cryptanalysis of the Full AES-192 and AES-256. In *Advances In Cryptology* (2009), pp. 1–18.

[5] Bogdanov A., Khovratovich D., R. C. Biclique Cryptanalysis of the Full AES. In *Advances In Cryptology* (2011), pp. 344–371.

[6] Brumley D., B. D. Remote Timing Attacks Are Practical. In *Proceedings of the 12th conference on USENIX Security Symposium* (2003), pp. 1–14.

[7] Chaum D. Blind Signatures for Untraceable Payments. In *Advances In Cryptology* (1983), pp. 343–356.

[8] Courtois N. All About Side Channel Attacks, 2013. Lecture Slides from COMPGA12 UCL – Applied Cryptography.

[9] Curtin M., D. J. A Brute Force Search of DES Keyspace. *;login: Magazine, vol. 23, no. 3* (1998).

[10] Daeman J., R. V. AES Proposal: Rijndael. Submitted to NIST at `https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf`, 1999.

[11] Daemen J. *Cipher and Hash Function Design Strategies Based on Linear and Differential Cryptanalysis.* PhD thesis, Katholieke Universiteit Leuven, 1995.

[12] Daemen J., R. V. *The Design of Rijndael.* Springer, 2002.

[13] Diffie W. Whitfield Diffie on NSA and Joseph Meyer Letter, 2010. `https://cryptome.org/0001/diffie-nsa.htm`, Last accessed on December 1, 2019.

[14] Diffie W., H. M. Exhaustive Cryptanalysis of the NBS Data Encryption Standard. *Computer, vol. 10, no. 6* (1977).

[15] Electronic Frontier Foundation. *Cracking DES.* O'Rielly and Associates Inc., 1998.

[16] Giacomin E., G. P. Differential Power Analysis Mitigation Technique Using Three-Independent-Gate Field Effect Transistors. In *IFIP International Conference on Very Large Scale Integration* (2018), pp. 107–112.

[17] Giraud C. DFA on AES. In *Lecture Notes in Computer Science, vol. 3373* (2004), AES '04, pp. 27–41.

[18] Hellman M., Merkle R., S. R., et al. Results of an Initial Attempt to Cryptanalyze the NBS Data Encryption Standard. Tech. rep., Stanford Electronics Laboratories, Standford University, 1976.

[19] Heys H. A Tutorial on Linear and Differential Cryptanalysis. *Cryptologia, vol. 26, no. 3* (2002), 189–221.

[20] Johnson T. *American Cryptography During the Cold War, 1945 – 1989; Book III: Retrenchment and Reform, 1972 – 1980.* National Security Agency, 2013.

[21] Kinnucan P. Data Encryption Gurus: Tuchman and Meyer. *Cryptologia, vol. 2, no. 4* (1978), 371–381.

[22] Mamiya H., Miyaji A., M. H. Efficient countermeasures against RPA, DPA, and SPA. In *Lecture Notes in Computer Science, vol. 3156* (2004), CHES '04, pp. 343–356.

[23] Mantin I. A Practical Attack on the Fixed RC4 in the WEP Mode. In *Advances In Cryptology* (2005), pp. 395–411.

[24] Matsui M. Linear Cryptanalysis Method for DES Cipher. In *Lecture Notes in Computer Science, vol. 765* (1994), EUROCRYPT '93, pp. 386–397.

[25] Murdock K., Oswald D., G. F., et al. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *Proceedings of the 41st IEEE Symposium on Security and Privacy* (2020), S&P '20.

[26] National Institute of Standards and Technology. Federal Information Processing Standards (FIPS) Activities. *Journal of Research of the National Institute of Standards and Technology, vol. 97, no. 6* (1992).

[27] National Institute of Standards and Technology. Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard. *US Federal Register, vol. 62, no. 177* (1997).

[28] National Institute of Standards and Technology. FIPS 46-3 – Data Encryption Standard, 1999. `https://csrc.nist.gov/publications/detail/fips/46/3/archive/1999-10-25`, Last accessed on November 15, 2019.

[29] National Institute of Standards and Technology. 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, vol. 104, no. 5* (1999).

[30] National Institute of Standards and Technology. Report on the Development of the Advanced Encryption Standard (AES). *Journal of Research of the National Institute of Standards and Technology, vol. 106, no. 3* (2000).

[31] National Institute of Standards and Technology. FIPS 197 – Advanced Encryption Standard (AES), 2001. `https://csrc.nist.gov/publications/detail/fips/197/final`, Last accessed on December 2, 2019.

[32] Schneier B. *Applied Cryptography*, second ed. John Wiley and Sons Inc., 1996.

[33] Tromer E., Osvik D., S. A. Efficient Cache Attacks on AES, and Countermeasures. In *Journal of Cryptology, vol. 23, no. 1* (2010), pp. 37–71.

[34] Wong W. Timing Attacks on RSA: Revealing Your Secrets Through the Fourth Dimension. *ACM Crossroads, vol. 11* (2005).