Ratsgymnasium Osnabrück

Facharbeit zum Rahmenthema
"Die Vermessung der Welt"

Schuljahr 2012/2013, C2-Profil

"Cryptography and Cryptanalysis in the Digital Age"

Verfasser:
Jan Toennemann

Fachlehrer:
Reinhard Flehr

Abgabetermin: 26.02.2013

Contents

1 Introduction

Nowadays, everything is tied to the Internet in some way. Tim Berners-Lee's invention, meant to ease the communication and collaboration inside the CERN research facility, is now a dominating part of the life of each and every one of us. Millions of people use it every day to transfer sensitive data, like doing bank transfers or sharing company documents. If a third party could obtain this information it would wreak havoc to the companies or individuals affected.

Because of this, there are security measures in place. In fact, they are used in way more areas than previously mentioned, ensuring safe and private browsing and communication. That is where the main subject of this paper comes in: Cryptography. Although cryptography was used long before the Internet, for example Caesar's use of the substitution cipher or the ingenious Enigma machine used by the Germans in World War II, it is now used all over the world.

Personally, I'm fascinated by technology in general, but cryptography is what intrigues me most at the moment. Current implementations combine mathematics and computer science in a unique way, especially the mechanics of asymmetric encryption, discussed in section 6, fascinate me.

Due to the wide distribution of cryptography I will need to strictly limit the topics covered, which is why I am focusing on the most frequently used implementations and a few older, simpler ones to begin on. I also had to leave out most of the mathematical part, because proving the security of each cipher covered in this paper would've raised the page count to around 30. Because of the technical nature of this subject I have focused on digitally published papers rather than printed books, which in addition results in enhanced availability of the cited material.

2 Preliminaries

2.1 XOR

XOR (exclusive or) is a logical operation performed on two operands that results in one if both values have opposite truth values, i.e. only one operand is true. In this paper, XOR operations will be denoted using the $\bigoplus$-operator. The following truth table should clarify the XOR operation:

Input_1	Input_2	Output	$(\text{Input}_1 \bigoplus \text{Input}_2)$
0	0	0	
0	1	1	
1	0	1	
1	1	0	

2.2 Sets

Finite sets will be denoted using $\{0,1\}^n$, with n being *the number of elements in the set* rather than a power, because we are in binary context. Further on, $|S|$ will denote the *size* of a set S, although in this case it equals n.

2.3 Functions

Functions map one or multiple inputs they receive as arguments to one or multiple outputs. They are deterministic, which means that when neither the input nor the function itself changes, the output will stay the same. To denote functions, the notation $f : X \to Y$ is used, with f being the function, X the input space and Y the output space.

2.4 Randomness

2.4.1 Random Variables

A random variable X is a function $X : S \to V$ with V being the set where the random variable takes it's values. For example, with $X : \{0,1\}^n \to \{0,1\}; X(y) = lsb(y) \in \{0,1\}$ (lsb is an operation extracting the *least significant bit(s)* of a binary number, the bits on the far right) and S being distributed uniformly, the probability of X being 0 is equal to the probability of it being 1.

2.4.2 Randomized Algorithms

While a deterministic algorithm $y \leftarrow A(m)$ always maps the point m to the *point* $A(m)$, a *randomized algorithm* $y \leftarrow A(m;r)$ with $r \xleftarrow{R} \{0,1\}^n$ (r is a random variable) maps the point m to the *output space* $A(m)$, thus resulting in multiple possible outputs for the same m depending on r.

2.5 Pseudo-Randomness

2.5.1 Pseudo-Randomness

Computers generally reside to Pseudo-Random Generators to achieve *random looking* output to use for various tasks rather than *truly random sequences*. "The reason lies in many extra benefits provided by pseudo-random generators. [...] [O]ne often needs to repeat the exact same sequence. With a truly random generator, one actually has to record all its outcomes: long and costly. The alternative is to generate pseudo-random strings from a short seed."[1] Those seeds play an important role in cryptography, because usually, *passwords*, *shared secrets* etc. will be used as seeds for the pseudo-random generator.

2.5.2 Pseudo-Random Generator

A pseudo-random generator expands a *short* input (*seed*) to a *large, seemingly random* output. Therefore, a PRG G with k as seed produces the output $G(k) \gg k$, which should be *indistinguishable* from $r \xleftarrow{R} \{0,1\}^n$.

2.5.3 Unpredictability

For a PRG to be unpredictable, there may not be a single *statistical test* that can distinguish the output of the PRG from the output of a truly random function.

[1] http://www.cs.bu.edu/fac/lnd/toc/z/node24.html

2.5.4 Pseudo-Random Functions

A pseudo-random function is a function defined over (K, X, Y) : $F : K \times X \rightarrow Y$ such that there exists an efficient algorithm to evaluate $F(k, x)$. *All* output from a PRF should be *indistinguishable* from random, regardless of the input, as long as the *function itself* was drawn at random.

2.5.5 Pseudo-Random Permutations

A pseudo-random permutation is a function defined over (K, X) : $E : K \times X \rightarrow X$ such that there exists an efficient algorithm to evaluate $E(k, x)$. The PRP $E(k, \cdot)$ is *one-to-one*, so there exists an effective inversion algorithm $D(k, y)$. A PRP should be *indistinguishable* from a truly random permutation.

2.6 Modular Arithmetic

2.6.1 Modular Arithmetic

"Modular arithmetic [...] is a system of arithmetic for integers, where numbers "wrap around" after they reach a certain value - the modulus"[2], here denoted as N. Although modular arithmetic is often denoted using *(mod n)* after an equation, here we will use *in* $\mathbb{Z}_N$ instead.

To explain modular arithmetic, the 12-hour clock is a splendid example. Basically, while a day has 24 hours, the clock can only display the hours one to twelve. Thus, on a 12-hour clock:
$14 \equiv 2$ in $\mathbb{Z}_{12}$; $17 \equiv 5$ in $\mathbb{Z}_{12}$; $23 \equiv 11$ in $\mathbb{Z}_{12}$ with $\equiv$ denoting a *congruence relation.*

2.6.2 Greatest Common Divisor

For a pair of integers a and b the *greatest common divisor* is denoted by $\gcd(a, b) = x$. If we use the integers 12 and 18 as example, we get $\gcd(12, 18) = 6$, because both 12 and 18 are divisible by 6, but not by any integer higher than that.

For every integer pair a, b there exists another integer pair x, y such that $x \times a + y \times b = \gcd(a, b)$. If $\gcd(a, b) = 1$ we say that a and b are *relatively prime.*

2.6.3 Modular Inversion

While inverting rational numbers is considered easy, for example the inverse of 2 being $\frac{1}{2}$, inverting integers in $\mathbb{Z}_N$ is considered to be very hard. To invert an element x in $\mathbb{Z}_N$ we need an element y in $\mathbb{Z}_N$ such that $x \times y = 1$ in $\mathbb{Z}_N$, in which case $y = x^{-1}$. For example, let N be an odd integer. Then the inverse of 2 in $\mathbb{Z}_N$ is $\frac{N+1}{2}$, because $2 \times (\frac{N+1}{2}) = N + 1 \equiv 1$ in $\mathbb{Z}_N$.

But beware, not every element in $\mathbb{Z}_N$ is invertible! For an element x in $\mathbb{Z}_N$ to be invertible, $\gcd(x, N) = 1$ must hold. We will call the *set of invertible elements* in $\mathbb{Z}_N$ from now on $(\mathbb{Z}_N)^*$, which basically means that $(\mathbb{Z}_N)^* = \{x \in \mathbb{Z}_N : \gcd(x, N) = 1\}$.

[2] http://math.wikia.com/wiki/Modular_arithmetic

2.6.4 Generators

The set of invertible elements $(\mathbb{Z}_N)^*$ is a *cyclic group*, which means that every element in $(\mathbb{Z}_N)^*$ can be generated from a single element g inside that group by raising g to a higher power, that is there exists a $g \in (\mathbb{Z}_N)^*$ such that $\{g^0, g^1, g^2, \ldots, g^{p-2}\} = (\mathbb{Z}_N)^*$. Any number inside $(\mathbb{Z}_N)^*$ which can be used as g is then called a *generator of* $(\mathbb{Z}_N)^*$ and the set $\{1, g, g^2, \ldots\}$ is called *the group generated by* g, denoted as <g>. If we use $N = 7$ and $g = 3 \in (\mathbb{Z}_N)^*$ as an example, <g>would be $\{3^0, 3^1, 3^2, 3^3, 3^4, 3^5\} = \{1, 3, 2, 6, 4, 5\}$.

Similar to modular inversions, one must watch out here, because not every element in $(\mathbb{Z}_N)^*$ can be used as a generator!

A shortcut to the size of the $(\mathbb{Z}_N)^*$ group is provided by Euler's totient function, which will be denoted as $\varphi(N)$, where $\varphi(N) = |(\mathbb{Z}_N)^*|$. When using the totient function on a multiple of two known primes, one can easy compute it using $\varphi(pq) = (p-1) \times (q-1)$.

2.6.5 Modular *e*'th Roots

With p being a prime and $c, e \in \mathbb{Z}_p$, $x^e \equiv c$ in $\mathbb{Z}_p$ is called an e'th root of c. For example, $7^{\frac{1}{3}} \equiv 6$ in $\mathbb{Z}_{11}$ because $6^3 = 216 \equiv 7$ in $\mathbb{Z}_{11}$.

Just like previously, we must pay attention, because not every e'th root exists. To determine the existence of an e'th root we have two options: The easiest of those would be to check if $\gcd(e, p-1) = 1$, because then for all $c \in \mathbb{Z}_p^*$, $c^{\frac{1}{e}}$ exists in $\mathbb{Z}_p$. The other case is for $e = 2$, which we will cover in the next subsection.

2.6.6 Square Roots (*e*'th Roots with *e*=2)

Should p be an odd prime, we would check for $\gcd(2, p-1) \neq 1$ to determine the existence of the *square root*. If it does exist and we succeed in finding x^2 in $\mathbb{Z}_p$, it could have originated from both x and $-x$ in $\mathbb{Z}_p$. All x in $\mathbb{Z}_p$ which have a square root in $\mathbb{Z}_p$ are called *quadratic residues*. When p is an odd prime, the number of QR's in $\mathbb{Z}_p$ is $(p-1)/2 + 1$.

2.6.7 Discrete Logarithm

For a prime $p > 2$ and g in $(\mathbb{Z}_p)^*$ consider the function $x \mapsto g^x$ in $\mathbb{Z}_p$. The inverse of that function would be the *discrete logarithm* of g on g^x, denoted by $\text{Dlog}_g(g^x) = x$. The following table uses $\mathbb{Z}_{11}$ as example:

x in $\mathbb{Z}_{11}$	1	2	3	4	5	6	7	8	9	10
$\text{Dlog}_2(x)$	0	1	8	2	4	9	7	3	6	5

To further illustrate this, here are some of them in detail:
$\text{Dlog}_2(5) = 4 \Leftrightarrow 2^4 = 16 \equiv 5$ in $\mathbb{Z}_{11}$, $\text{Dlog}_2(8) = 3 \Leftrightarrow 2^8 = 256 \equiv 3$ in $\mathbb{Z}_{11}$, $\text{Dlog}_2(1) = 0 \Leftrightarrow 2^0 = 1 \equiv 1$ in $\mathbb{Z}_{11}$.

The discrete logarithm is considered to be a hard problem, because currently no efficient algorithm can compute the discrete logarithm in large $(\mathbb{Z}_p)^*$ in short time.

2.6.8 Trapdoor Functions

"Trapdoor functions are (injective) functions that are easy to evaluate but hard to invert unless given an additional input called the trapdoor. Specifically, the classical security notion considered for trapdoor functions is one-wayness, which asks that it be hard to invert (except with very small probability) a uniformly random point in the range without the trapdoor."[3] Further explanation will take place in section 6.2.

3 Stream Ciphers

3.1 General

Stream ciphers are *symmetric key ciphers* (the same key is used for both *encryption* and *decryption*) that combine a message given as a *plaintext* with a *keystream*, which is pseudorandomly generated from an input key, to a *ciphertext*. Although there are many ways to combine the message with the keystream, in digital cryptography this is usually done using XOR-operations. Starting with the simplest stream cipher available in section 3.2, we will also discuss Salsa20 later on in section 3.3, although not as detailed as the first one.

3.2 The One-Time Pad

3.2.1 The One-Time Pad (1927, Gilbert Vernam)

The one-time pad is a very simple cryptosystem in which a message is "masked" using a chunk of data *of the same length*, resulting in a ciphertext. It is one of the simplest ciphers available, very easy to apply in digital environments and can under certain circumstances even be *perfectly secure*. The downside is that the aforementioned certain circumstances are the strictest possible: as indicated by the name, the key *must not be reused*, or the resulting ciphertexts will be fairly easy to crack.

Let $\mathbb{M}$ and be the message space and $\mathbb{C}$ be the ciphertext space over $\{0,1\}^n$.
Let $\mathbb{K}$ be the key space over $\{0,1\}^n$. Let $m \in \mathbb{M}$ be the message.
Let k be the key, randomly distributed over $\mathbb{K}$.
The encryption of m under the key k is $E(k,m) = m \bigoplus k = c$.

Figure 1: Theoretical implementation of the OTP in a binary environment.

3.2.2 Using the OTP As a Stream Cipher

In order to use the one-time pad as a stream cipher, we will need to replace the *random* key by a *pseudorandom* one, for which we will use an unpredictable pseudo random generator. This will allow us to extend a small seed to a seemingly random sequence long enough to act as a key for the OTP, regardless of the message length.

[3] O'Neill, Adam, Stronger Security Notions for Trapdoor Functions and Applications, Atlanta 2010

Let $G(k)$ be the key generated by the PRG G under the seed k.
The encryption of m under the seed k is $E(k, m) = m \bigoplus G(k) = c$.

Figure 2: Theoretical implementation of the OTP as a stream cipher.

3.2.3 Main Flaw of the OTP

As already mentioned in section 3.2.1, re-use of the key makes it possible to crack the OTP with ease. Due to the message simply being XOR'd with the key, multiple ciphertexts created with *different messages* but *the same key* share the relationship indicated in figure 3.

Let $m_1 \in \mathbb{M}$ and $m_2 \in \mathbb{M}$ be messages where $m_1 \neq m_2$.
$c_1 \leftarrow m_1 \bigoplus G(k)$, $c_2 \leftarrow m_2 \bigoplus G(k)$
$c_1 \bigoplus c_2 = m_1 \bigoplus m_2$

Figure 3: Relationship of OTP-encrypted messages that share the same key.

Therefore, with enough ciphertext pairs given, one can obtain the key by comparing the XOR-values with language-specific letter redundancy data. This flaw is the main reason for the one-time pad to be considered generally impractical. Also, please note that the OTP in it's current state does not protect against unwanted changes, which can occur due to an attacker, because of communication errors, through drive corruption and so forth.

3.3 Salsa20

3.3.1 Salsa20 (2005, Daniel J. Bernstein)

Part of the eSTREAM portfolio, Salsa20 is a family of stream cipher algorithms by Daniel J. Bernstein. Snuffle 2005 is the encryption function of that family and it's using "a strong cryptographic hash function [...] to efficiently encrypt data"[4]. In its entirety, the Salsa20 algorithm family consists of 3 main functions (for *hashing*, *expanding* and *encrypting* input) with some of them depending on smaller, custom implementations done by Bernstein. Discussing all algorithms in detail would go beyond the scope of this paper, for that see source 33. The encryption function itself uses a long chain of simple operations to effectively encrypt given data, which can be generalized as *32-bit addition*, *32-bit exclusive-or* and *constant-distance 32-bit rotation* (also called *byte shifting*).

When using the primary variation of Salsa20 one needs to supply a *256-bit key* and a *64-bit nonce*. A nonce is typically a short sequence which is required to be unique to guarantee safety. Often, a counter that is present on both the senders and the receivers side and increasing every time a message gets exchanged is used as a nonce, as it's value does not need to be transmitted along with the message and is generally considered to be safe enough. The given input gets expanded into a 2^{70}-byte stream and XOR'd with the first b bytes of the message.

Unlike other popular stream ciphers Salsa20 does not incorporate the plaintext of the message in any way, so the resulting ciphertext is entirely independent of the message supplied. Another feature of this cipher is the generation

[4]Bernstein, Daniel J., Salsa20 design, Chicago 2005

of a block-divided output stream, similar to block ciphers discussed in the next section, which allows for random reads in the output stream and - more important - for *parallelism*.

3.3.2 Parallel Computing

With multi-core processors being a standard for multiple years now, parallel computing has become an important method to achieve a better performance. Basically, the task (in this case "encrypt message m under the key k and nonce n") is split into smaller tasks (i.E. "encrypt message $m[0-4]$ under the key k and nonce n", "encrypt message $m[5-9]$ under the key k and nonce n", etc.) and handled by each of the CPU cores independently. Using synchronization, the results of the tasks will be merged into the final output stream.

But for ciphers to allow for parallel execution there are certain design specifics to keep in mind, some of which bear negative sides. The method used here is the most widespread one: dividing the output into blocks achieves concurrency easily, but produces some overhead in the output stream as the hash of both the key and the nonce need to be present in each block of the output stream.

4 Block Ciphers

4.1 General

Block ciphers are symmetric key ciphers that operate on fixed-size blocks (specified by the cipher's *blocksize*) of plaintext and output the ciphertext equivalent of that block under a supplied key. For block ciphers it is possible to run multiple operations on one block of ciphertext, resulting in a variety of combinations arising from this, some of which are covered later in this section. Starting with a now deprecated block cipher, the previous NIST (National Institute of Standards and Technology) standard DES which uses a *Feistel network*, we'll continue with the current NIST standard AES, which takes another, fairly different approach on applying several operations to the blocks.

4.2 Data Encryption Standard

4.2.1 Data Encryption Standard (1977, IBM)

Once a federal standard, the Data Encryption Standard, abbreviated *DES*, was developed in the 70s at IBM based on the work of Horst Feistel, which is why the main function of the cipher is called the *Feistel function*. DES originated from IBM's *Lucifer* cipher which was slightly modified by the U.S. National Bureau of Standards and released under it's new name. Due to those changes being minimal, DES is considered to be a different version of Lucifer rather than it's successor and thus considered to be the first publicly available block cipher.

Please note that DES is already deprecated and not considered safe anymore. The Electronic Frontier Foundation showed 1998 that DES could be broken with a simple *brute force attack*, like described in section 4.2.4 and shown in section 4.2.5. While that was the most popular cracking of DES, one year earlier

the DESCHALL Project already succeeded in decrypting a DES-encrypted ciphertext with an unknown key using thousands of computers connected via the Internet as part of a contest series sponsored by RSA Security. Incipient stages of DES cracking have existed since the late 70s, but those were just not feasible at that time, because it would've needed several million dollars to be carried out.

4.2.2 DES Core - the Feistel Network

The Data Encryption Standard is considered to be a typical *Feistel cipher*, because it follows a scheme designed by the German cryptographer and physicist Horst Feistel, which is called the *Feistel network*. When embarking on a plaintext block, the 64-bit input, consisting of 56 bits of the message and 8 bits for parity checking, is split in two 32-bit halves and each half gets forwarded into the *initial permutation*. When encrypting, the two permuted halves *R0* and *L0* are then processed like shown in figure 4, with the 16 iterations (often called *rounds*) being identical processing stages. One iteration consists of the "upper" half being transformed by the Feistel function and then XOR'd with the "lower" half. Along with the half block a *subkey*, derived from the key given to the cipher, is given to the Feistel function. The subkey derivation process is quite long, for now it's enough to say that the main key runs through a permutation and then enters a phase similar to the Feistel network. Apart from the last iteration, after the XOR-operation took place, the halves swap places, which means that the previously "lower" half is now on top and vice versa. After all iterations, the final permutation is applied to the block halves called "L 16" and "R 16" in figure 4, which are then merged together and form the ciphertext.

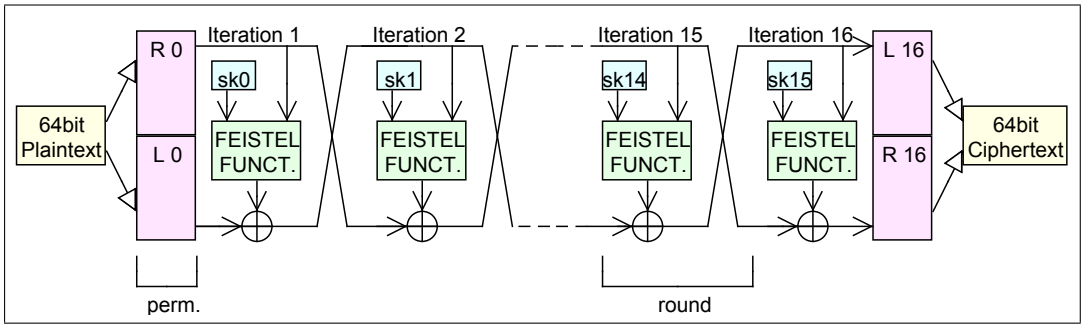


Figure 4: Encryption using the Feistel Network in DES

To decrypt a DES-encrypted ciphertext, one simply needs to run the ciphertext through the Feistel Network again, but swap the halves before the initial permutation and apply the subkeys in reverse order, as indicated in figure 5.

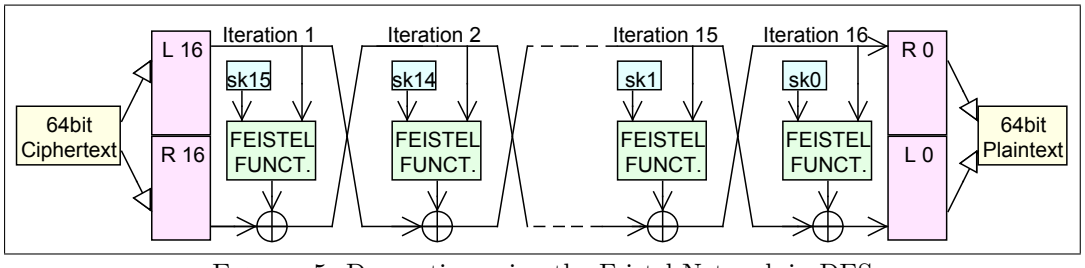


Figure 5: Decryption using the Feistel Network in DES

4.2.3 The Feistel Function in DES

Unique to each implementation of the Feistel network is the Feistel function which operates on a half block and a subkey. The length of each subkey in DES is 48 bits, which means that the 32-bit input, the half block given to the Feistel function, needs to be expanded to 48 bits to continue processing. This is done using DES' *expansion permutation*, which duplicates some bits in order to reach the desired length. The - now 48-bit long - input half-block will then be XOR'd with the subkey. Then, the data is split into 8×6-bit pieces which will be processed by 8 separate *substitution boxes* (*S-boxes*), mapping the 6 input bits to 4 output bits, each box using it's own specific *lookup table* (similar to a ruleset) to determine the output. Thus, each S-box is a different function $\{0,1\}^6 \rightarrow \{0,1\}^4$. The S-boxes are what provide the security of DES - trying to reverse the look-up without knowing the key would result in a tremendous amount of combinations. After going through the S-boxes, the 32 output bits of these will be merged using a *fixed permutation*, called the *P-box*.

4.2.4 Exhaustive Search Attacks

Limiting the scope to cryptography, *exhaustive search* attacks are a range of attacks on ciphers that are as simple as possible - they don't exploit specific weaknesses or utilize known connections between ciphertexts and/or plaintexts. The most well known exhaustive search attack is the *brute force* attack, which simply tries *all available keys* on a cipher until it has found the correct one. While this will definitely yield a correct result, it often just isn't feasible with current hardware, because it would take too much time. The time required is linked to the cipher strength - in the case of block ciphers it is determined by the block size.

For example, a message encrypted with 256-bit AES has 1.1×10^{77} possible keys, trying to crack it with the currently fastest supercomputer, which is, according to source 46, capable of 17.59 Petaflops (quadrillions of floating point calculations per second), when assuming 1500 Flops are required to try a key (which is quite optimistic), it would take $(1.1 \times 10^{77}) / \frac{(17.59 \times 10^{15})}{1500}$ seconds to crack a key. This equals to roughly 9.38×10^{63} seconds, around 1.09×10^{59} days, approximately 2.97×10^{56} years or - frankly - 297 *septendecillion* years.

Other exhaustive search attacks are *dictionary attacks*, which work similar to brute force attacks but limit the combinations they try on combinations of words that they look up in prepared files, often extracted from dictionaries. While dictionary attacks are way faster than normal brute force attacks, there is a quite big possibility that they won't yield any results and their overall running time is usually still way too long to be worthwhile.

4.2.5 Exhaustive Search Attack on DES

Although generally ciphers are secure enough to withstand exhaustive search attacks, as shown in the previous section, DES is a fine example for one that isn't - *anymore*. When DES was released to the public in 1977 it certainly met the standards, there was no feasible attack in sight. But as time passed, technology improved, and it was easier to encrypt and decrypt with DES. While this speed benefit was good for the computer industry, it increased way faster

than predicted at the time DES was developed. The cipher strength of DES is - in fact - not 2^{64} as one might think because of it's block size, but rather 2^{56} because of the eight bits used for parity checking as mentioned in section 4.2.2. This results in "only" 7.2×10^{16} possible keys for DES, which is *extremely* low in comparison to the amount used for the example in section 4.2.4.

If we do the calculation again, using the possible combinations for DES instead of the AES values, we see that it would take $(7.2 \times 10^{16})/\frac{(17.59 \times 10^{15})}{100}$ seconds to crack a key. Please note that the Flops required to try a key has been lowered to a way smaller estimate because of the simpler cipher used. Using the Titan to crack a DES key would take merely 409 seconds, which is a little less than 7 minutes.

This result means that it would even be possible for a small network of consumer range computers with modern hardware to crack a DES key in some weeks, which clearly makes it unsuitable for anything remotely confidential.

The machine built by the EFF in 1998 used $1,536$ custom-built chips and was able to crack a DES key in a couple of days, which in turn resulted in the search for a successor. This successor was found in the Advanced Encryption Standard, discussed in the next section.

4.3 Advanced Encryption Standard

4.3.1 Advanced Encryption Standard (1998, Vincent Rijmen and Joan Daemen)

Originally called *Rijndael*, the Advanced Encryption Standard is the follow-up NIST specification for encrypting electronic data since 2001 and thus considered to be the successor of DES, although AES takes a completely different approach as a block cipher. This cipher was developed by the Belgian cryptographers Vincent Rijmen and Joan Daemen. First published in 1998, it is now used all over the world in many different applications. Similar to its predecessor, the standard is a slightly modified version of the submitted cipher, in this case the block size has been fixed to 128 bits and the possible key sizes have been limited to 128, 192 or 256 bits.

4.3.2 The Substitution-Permutation Network

Unlike DES, the AES cipher does not use a Feistel network but rather a structure called *substitution-permutation network* (SPN) which heavily utilizes S-boxes and a P-box to create the ciphertext. In fact, due to the S-box usage of DES, one can compare the SPN with the Feistel network version DES uses.

Round keys (similar to subkeys) are generated from the main key and XOR'd with the current AES *state* before each round once after the last one (see section 4.3.3 for details on the SPN in AES). A round resembles the Feistel function in DES: The input split up and gets modified by some S-boxes, which are functions $\{0,1\}^4 \to \{0,1\}^4$ this time, and by a P-box, which is omitted in the last round.

To decrypt a ciphertext that has been encrypted using a SPN, one simply needs to apply the inverses of the S- and P-boxes and apply the round keys in reverse order.

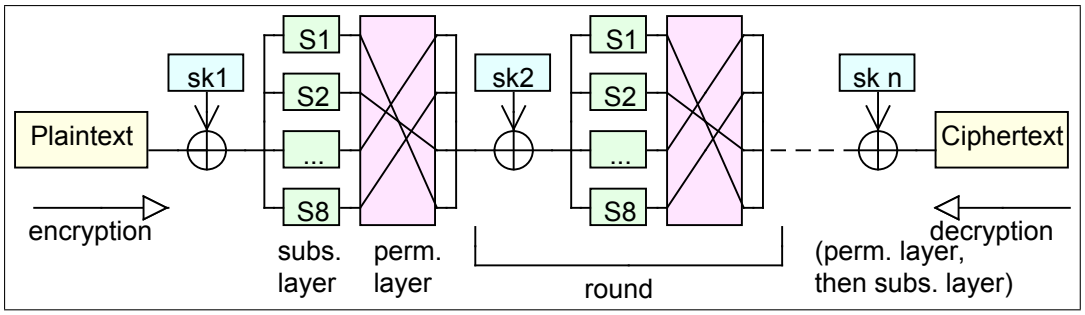


Figure 6: Encryption and decryption in the SPN

4.3.3 How AES Operates

All versions of Rijndael and thus AES operate on a column-major ordered byte-matrix, like illustrated in figure 7. The number of iterations is determined by the key size chosen. For an 128-bit key, AES will use 10 rounds, for an 192-bit key 12 rounds and if using an 256-bit key, it will use 14 rounds to process each input block. A round in AES is rather complex in comparison to what we have

$$\begin{bmatrix} 6 & 3 & 1 & 7 \\ 1 & 9 & 5 & 0 \\ 1 & 0 & 2 & 4 \\ 2 & 4 & 5 & 9 \end{bmatrix} \quad \begin{bmatrix} 3 & 4 & 2 & 1 \\ 9 & 7 & 5 & 3 \\ 2 & 0 & 3 & 8 \\ 6 & 0 & 5 & 2 \end{bmatrix}$$

6	1	1	2	3	9	0	4	1	5	2	5	7	0	4	9
3	9	2	6	4	7	0	0	2	5	3	5	1	3	8	2

when saved contiguously in memory.

Figure 7: Examples of AES states and their memory transcription.

already seen, consisting of several processing steps unique to AES, independent of each other, but dependant on the key.

In the first round, the current state gets XOR'd with the round key. This operations is typically referred to as the *AddRoundKey* step and constitutes the last operation of each following round.

All following rounds - excluding the final round - consist of 4 different operations: First of all, there is the *SubBytes* step, where each byte in the state is run through a S-box and is replaced by the output. After that, the *ShiftRows* step is applied to the current state, shifting each value in the second row of the state one to the left, the ones in the third row two to the left and the each value in the last row three to the left (= one to the right). The last new operation applied to the state is the *MixColumns* step, which applies fixed operations on each byte in the column it's currently processing, with each of the four input bytes severely effecting the output. These three steps are followed by the already known *AddRoundKey* step.

The last iteration is almost like the preceding ones, but there is one major difference: the *MixColumns* step is omitted.

For illustrations on the operations used in AES please refer to appendix figure 1.

5 Message Authentication Codes

5.1 General

After discussing ciphers in the last two major sections it's time to widen the scope a bit. There is more to digital cryptography than just encryption and decryption, which is supposed to provide *confidentiality*.

The topic discussed in this section is about the Message Authentication Codes (MACs), which are appended as a tag to an encrypted ciphertext to provide *integrity*. By executing the same algorithm on the received message as used by the sender, one can compare the own results with the tag appended to the message, which is how errors that occurred during transmission of the data over a network or attempts of adversaries to alter ciphertexts can be detected.

Due to the limited extent of this paper I decided to only present one example of a MAC algorithm, which is built on the foundation provided by block ciphers. However, please keep in mind that there are a lot of very different MAC algorithm implementations out there that differ severely in some cases.

5.2 Cipher-based MAC

5.2.1 Cipher-based MAC (2003, Tetsu Iwata and Kaoru Kurosawa)

The cipher-based MAC (CMAC) is a message authentication code algorithm based on the concept of block ciphers. CMAC is a derivative of CBC-MAC (general specification on MAC algorithms utilizing block ciphers) and was recommended as a "block cipher mode of operation" by the NIST in 2005. Similar to previous cases, the version published by the NIST is a renamed version of a submission, in this case the cipher was called OMAC1 prior to the recommendation.

5.2.2 CMAC Tag Generation

Like all CBC-MACs, CMAC is cipher-dependant, which means that the output of the algorithm is determined by the cipher used. The usual choice for CMAC is AES-128, but to keep this more general we will consider any b-bit block cipher E. Requirements for all MAC algorithms are a secret key, which is usually forwarded to the cipher used, and a value l denoting the desired length of the tag, and, of course, an input message to generate the tag for. This input message can either be a plaintext or a ciphertext, although generating a tag on the ciphertext is more common, as it saves the time to decrypt the ciphertext when anomalies are detected.

CMAC starts out by generating two extra keys (k_1 and k_2), incorporating the block cipher and a constant based on it's blocksize. One of these extra keys will be used on the last message block, which one will be decided by the length of the last block: If it is a full block ($|m[n]| = b$) then k_1 will be used, otherwise it's k_2. By now you might have noticed that the message is split into blocks again - a speciality of CBC-MAC. Then CMAC fixes a variable, c_0, to 0^b, so there are as many zeros as there are bits in a block. After having the initial value for c_0, the algorithm enters a loop and calculates the recursive function $c_i = E_k(c_{i-1} \bigoplus m_i)$ for $i = 1, ..., n$. The final tag will then be $msb_l(c_n)$, with

msb being the function outputting the most significant bits (the ones to the far left).

A simplified version of this process is illustrated in figure 8, think of $F(k, \cdot)$ as the previously mentioned recursive equation.

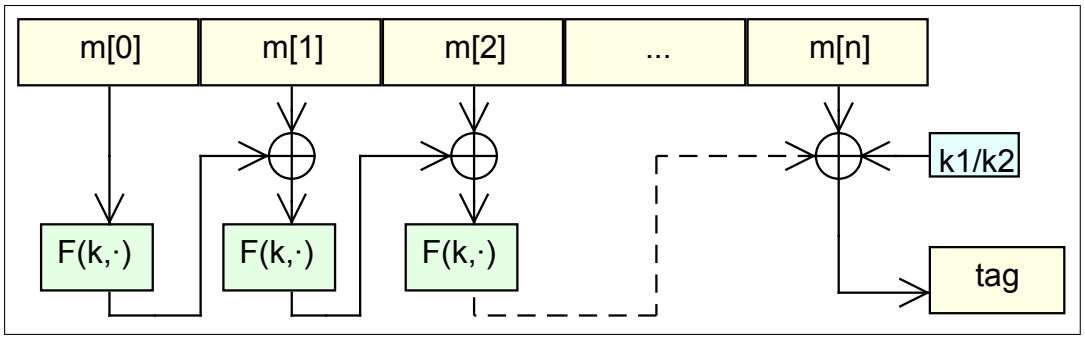


Figure 8: Tag generation in CMAC

6 Public-Key Cryptography

6.1 General

In the next-to-last section of this document, I'd like to introduce you to the concept of public-key cryptography which takes a different approach on encryption and decryption compared to the symmetric key ciphers discussed previously. The main difference is that public-key ciphers do not use the same key to encrypt and decrypt but rather two different ones, which declares them as *asymmetric key ciphers*. What has been encrypted with the first key in the key pair can only be decrypted with the second one in that pair and vice versa. Because of this, one of the keys is often available publicly to allow for encrypted communication between two parties without having to share the secret key first.

For example, if two people want to exchange encrypted mails, they both publish their second (public) key to a server and then each person is able to encrypt their message using the public key of the other, who will decrypt it upon receipt using their first (private) key. As explained in detail in section 6.2.2, the size of the plaintext is extremely limited, which is why public-key ciphers are usually used to securely transmit a session key for symmetric ciphers.

The asymmetric public-key cipher discussed in the following section is the most common cipher used in the public-key cryptography domain and is still considered to be secure, despite it's age.

6.2 RSA

6.2.1 RSA (1977, Ron Rivest, Adi Shamir and Leonard Adleman)

The RSA algorithm was the first publicly available usable asymmetric key cipher, which uses the factoring of large integers as integral problem to provide the security base. It's name arose from the initials of it's authors surnames Ron Rivest, Adi Shamir, and Leonard Adleman, which were working together

at the MIT back then. A similar algorithm was already invented in 1973 by Clifford Cocks in the UK, but was kept secret until 1997.

Because it plays the most important role in understanding RSA and the way it works, we will discuss key generation in detail in this paper. This section will make heavy use of the knowledge provided in section 2.6, so if you should struggle following through with the mathematics, consider revising that section.

6.2.2 RSA's Asymmetric Key Generation

To generate an asymmetric key pair (1024 bit in this case), the generator chooses two prime numbers p and q at random, each of a length of around 1024 bits. For comparison, the largest integer that would fit into 1024 bits would be roughly 3.60×10^{308}. Then, those two primes are multiplied and the result is saved as N. Again, to illustrate the number range we are discussing: as p and q are around 1024 bits, we would get $(2^{1025} - 1)^2$ by multiplying them together, which correlates to around 1.29×10^{617} as an integer.

Next up, the generator chooses two integers e and d, one being the "encryption" and one the "decryption" exponent, such that $e \times d \equiv 1$ in $\mathbb{Z}_{\varphi(N)}$. Usually, the desired length for e is quite short, but not too short in order to maintain security. The most common value for e is actually $2^{16} + 1$, which equals $65,537$. Despite the names, please keep in mind that ciphertext encrypted with the decryption key can be decrypted with the encryption key, too.

The final public key will then be the pair (N, e), while the secret key will consist of (N, d).

6.2.3 Encryption and Decryption in RSA

Converting plaintexts to ciphertexts and vice versa is actually quite simple when the key generation process was understood.

When encrypting a message m, one needs to turn it into an integer in the range $0 \leq m_{int} < N$ using a reversible *padding scheme*. There are a lot of padding schemes available and - as the name suggests - they raise the size of the message quite a bit, but provide additional security to make up for it. The recommended padding scheme for RSA is the *Optimal Asymmetric Encryption Padding* (*OAEP*), which is based on a Feistel network. After having converted the message into it's padded integer form, one only needs to calculate $F(m, e) = {m_{int}}^e$ in $\mathbb{Z}_N$ to get the ciphertext c.

When decrypting a ciphertext c, one needs to recover m_{int} from c and reverse the padding scheme. In order to get the padded message out of the ciphertext, the receiver only needs to calculate $F(m, e)^{-1} = F(m, d) = c^d$ in $\mathbb{Z}_N$. Now, having m_{int} and knowledge about the padding scheme used (which is usually standardized), he is able to undo the padding and receive m.

Please note that using the RSA cipher for encryption and especially decryption is computationally expensive in comparison to symmetric ciphers. Also, the size of the data that can be encrypted is severely limited. To calculate the maximum size of bytes one can encrypt, we calculate $\frac{|k|}{8}$ or $\frac{|k|-384}{8} + 7$ when using OAEP.

6.2.4 Why the RSA Algorithm is a Trapdoor Function

As mentioned in section 2.6.8, a trapdoor function is considered to be one-way. In the case of RSA this is accomplished by the Dlog-problem, which is also called the *RSA problem*. Because of this RSA is considered secure until it is possible to calculate the discrete logarithm of a huge number in efficient time, which would reveal the secret exponent d when only possessing knowledge of e and N.

7 Measurements

7.1 General

All following tests have been conducted using a pre-compiled binary of the program shown in appendix listing 1, which I named *CryptoBench*, simply because it's a benchmark based on cryptography. The devices used were my PC with both Windows 7 (64bit) and Linux (CrunchBang Waldorf 20130119 64bit) as operating system and with an *AMD Phenom II X4 965* at *3.4GHz core speed* as CPU and my Laptop, with Linux (CrunchBang Waldorf 20130119 64bit) as operating system and an *Intel Core i5-560M* at *2.66GHz core speed*.

Apart from the OTP and Salsa20, all ciphers were already implemented in the standard libraries of the Go programming language, which made them easier to use and better optimized than I could. Luckily, Salsa20 was already a testing candidate for the standard libraries, which made it possible to obtain a semi-official implementation including hardware optimization. Thus, the OTP is the only cipher that has been completely implemented by me and which did not receive any kind of optimization targeted to CPU architectures.

The following tables have been coloured to facilitate comparison, with the colour choices being rather straightforward: Green-colored cells contain the best time in the row, yellow-colored ones the intermediate one and red-colored ones the worst measured time.

7.2 Results

Results Table 1: One-Time Pad encryption (equals decryption); message m, $|k| = |m|$

	PC (Windows)	PC (Linux)	Laptop (Linux)
1024kb m	3ms	3ms	3ms
2048kb m	7ms	6ms	7ms
4096kb m	13ms	11ms	16ms
8192kb m	62ms	30ms	25ms
16384kb m	125ms	58ms	50ms

These numbers are not really surprising, OTP encryption consists of a simple XOR operation and is extremely fast. What's standing out is that - on larger input - the Windows test takes twice as long as the two on Linux. The most likely explanation for this is that - although all test systems have been stripped down to a bare minimum during runtime - the Windows environment

Results Table 2: Salsa20 encryption (equals decryption); message m, 256bit key k, nonce n

	PC (Windows)	PC (Linux)	Laptop (Linux)
1024kb m, 64bit n	2ms	1ms	1ms
2048kb m, 64bit n	3ms	3ms	2ms
4096kb m, 64bit n	7ms	6ms	5ms
8192kb m, 64bit n	16ms	17ms	14ms
16384kb m, 64bit n	32ms	31ms	23ms
4096kb m, 192bit n	7ms	6ms	5ms
8192kb m, 192bit n	15ms	14ms	11ms
16384kb m, 192bit n	29ms	28ms	22ms

Results Table 3: DES encryption (E :) and decryption (D :); message m, 64bit key k

	PC (Windows)	PC (Linux)	Laptop (Linux)
E : 1024kb m, 131072 rounds	873ms	834ms	864ms
E : 2048kb m, 262144 rounds	1760ms	1669ms	1729ms
E : 4096kb m, 524288 rounds	3489ms	3342ms	3442ms
E : 8192kb m, 1048576 rounds	6943ms	6681ms	6867ms
E : 16384kb m, 2097152 rounds	13865ms	13363ms	13855ms
D : 1024kb m, 131072 rounds	862ms	832ms	868ms
D : 2048kb m, 262144 rounds	1731ms	1665ms	1725ms
D : 4096kb m, 524288 rounds	3448ms	3329ms	3472ms
D : 8192kb m, 1048576 rounds	6903ms	6660ms	6930ms
D : 16384kb m, 2097152 rounds	13791ms	13353ms	13790ms

was still a bit more bloated and more CPU power was consumed by the OS itself.

When looking at results table 2 in comparison to results table 1 we notice something odd: although Salsa20 is way more complex than the OTP, it required *less* time. In fact, it's more than twice as fast, on the Windows system even four times faster than the OTP. This is due to the hardware optimization mentioned in section 7.1, which makes Windows a worthy competitor again. Another thing we can see is that the hardware optimization for the operations performed by Salsa20 is better on the Intel processor in the Laptop than on the AMD processor in the PC, as the Laptop tops the PC in every Salsa20 test, although it's core runs at fewer Gigahertz.

Compared to the previous two tables, results table 3 makes DES seem *extremely* slow in both encryption and decryption. And, well, that is exactly the case. Because DES was deprecated for a long time now, hardware support has been removed since generations of CPUs and all testing devices are relatively new. Thus, none of them have DES-specific optimization instructions built in, which makes this test a competition of raw calculation power, which is why the PC with Linux wins every round. When using Windows, the PC is almost equal to the Laptop, sometimes better and sometimes worse.

The AES cipher is presenting us a way superior display in results table 4: Although the operations are more complicated than the ones used in DES, it is faster by a factor of around 50 to 70 *times*! This is not just a side effect of

Results Table 4: AES encryption (E :) and decryption (D :); message m, key k

	PC (Windows)	PC (Linux)	Laptop (Linux)
E : 1024kb m, 128bit k, 65536 rds.	13ms	12ms	12ms
E : 2048kb m, 128bit k, 131072 rds.	26ms	24ms	25ms
E : 4096kb m, 128bit k, 262144 rds.	51ms	49ms	49ms
E : 8192kb m, 128bit k, 524288 rds.	101ms	98ms	99ms
E : 16384kb m, 128bit k, 1048576 rds.	203ms	199ms	198ms
E : 4096kb m, 192bit k, 262144 rds.	58ms	55ms	59ms
E : 8192kb m, 192bit k, 524288 rds.	116ms	111ms	115ms
E : 16384kb m, 192bit k, 1048576 rds.	231ms	222ms	227ms
E : 8192kb m, 256bit k, 524288 rds.	130ms	125ms	126ms
E : 16384kb m, 256bit k, 1048576 rds.	260ms	248ms	256ms
D : 1024kb m, 128bit k, 65536 rds.	12ms	12ms	12ms
D : 2048kb m, 128bit k, 131072 rds.	27ms	24ms	24ms
D : 4096kb m, 128bit k, 262144 rds.	50ms	49ms	48ms
D : 8192kb m, 128bit k, 524288 rds.	102ms	99ms	97ms
D : 16384kb m, 128bit k, 1048576 rds.	206ms	198ms	200ms
D : 4096kb m, 192bit k, 262144 rds.	57ms	55ms	56ms
D : 8192kb m, 192bit k, 524288 rds.	119ms	112ms	113ms
D : 16384kb m, 192bit k, 1048576 rds.	232ms	223ms	224ms
D : 8192kb m, 256bit k, 524288 rds.	130ms	125ms	125ms
D : 16384kb m, 256bit k, 1048576 rds.	262ms	250ms	250ms

the halved round count, but mainly due to AES being a NIST standard and thus having specific, low-level instructions on every CPU to severely accelerate the encryption and decryption process. No operating system really sticks out here, and although Windows is almost always slower than Linux, it does not perform too bad. Changing the key size from 128bit to 192bit doesn't slow the cipher down too much, changing it to 256bit then shows a slowdown of around 20% compared to 128bit, which isn't as much as one might have expected.

Now results table 5, the RSA table, is unique and not really comparable to the previous. On the one hand, we have the drastic reduction in the message length - while the other started with 1024kb (8388608bit), the first RSA encryption works on 96bit, which is due to the length limitation discussed in section 6.2.3. On the other hand, the encryption and decryption processes are different and there is a rather long span between encryption and decryption times.

Starting with the key generation, we can see that larger keys mean an exponential increase in generation time. However, the key generation is based on a RNG, which means that the speed of the key generation is "luck"-based, as in how fast the algorithm can find a suitable number in the output of the RNG, which is why the numbers are not really comparable.

The encryption process however is extremely fast, which is not too surprising, as the messages are quite small. Due to Windows not being able to measure time in any unit smaller than milliseconds, I built a workaround into the program to show that the measured time was less than one millisecond. But as we can't be sure which time exactly this was, it makes the Windows PC automatically

Results Table 5: RSA key generation (G :), encryption (E :), decryption (D :); message m, key k

	PC (Windows)	PC (Linux)	Laptop (Linux)
G : 1024bit k	118ms	151ms	162ms
G : 2048bit k	1437ms	1145ms	1762ms
G : 4096bit k	7301ms	25775ms	11430ms
G : 8192bit k	59936ms	215s	328s
G : 16384bit k	2367s	909s	1928s
E : 96bit m, 1024bit k	$<$ 1ms	92μs	94μs
E : 128bit m, 2048bit k	$<$ 1ms	182μs	219μs
E : 192bit m, 4096bit k	1ms	495μs	609μs
E : 256bit m, 8192bit k	2ms	1ms	1ms
E : 384bit m, 16384bit k	5ms	5ms	6ms
D : 96bit m, 1024bit k	4ms	4ms	4ms
D : 128bit m, 2048bit k	21ms	19ms	20ms
D : 192bit m, 4096bit k	104ms	98ms	109ms
D : 256bit m, 8192bit k	558ms	540ms	646ms
D : 384bit m, 16384bit k	3379ms	3530ms	4055ms

lose these rounds.

In contrast to the encryption process, the decryption process is pretty slow compared to the message length. Just like with the key generation, we can see an exponential rise in the time, related to the key length.

7.3 Conclusion

After discussing both symmetric and asymmetric ciphers and seeing how they perform, it's time to draw a conclusion. It should be obvious that neither the two symmetric cipher types, stream ciphers and block ciphers, nor any of them and the asymmetric ciphers are indeed comparable, as their use cases vary too much.

Generally said, keeping to current standards ensures confidentiality and efficiency, but widening the scope a bit to discover alternatives like Salsa20 can also pay off, as we can see an obvious speed benefit. In the end, it's always the choice of the user, and because the cryptographic community is so big and experienced, possible errors and weaknesses in ciphers are pointed out very fast, so that with a little bit of research, one can avoid trouble.

Another thing I want to emphasize is that under *no circumstances*, one should try to invent his own cipher. There are a lot of examples where people and companies did exactly that and it backfired completely. *Always* use pre-defined, public, thoroughly tested ciphers made by cryptographers, where possible weaknesses will be exposed by experienced people.

Sources

1. Boneh, Dan, Information theoretic security and the one time pad (19 min), Introduction to Cryptography, Stanford Online
2. Boneh, Dan, Stream ciphers and pseudo random generators (20 min), Introduction to Cryptography, Stanford Online
3. Boneh, Dan, Review PRPs and PRFs (12 min), Introduction to Cryptography, Stanford Online
4. Boneh, Dan, Public-key encryption (11 min), Introduction to Cryptography, Stanford Online
5. Boneh, Dan, Notation (15 min), Introduction to Cryptography, Stanford Online
6. Boneh, Dan, Fermat and Euler (18 min), Introduction to Cryptography, Stanford Online
7. Boneh, Dan, Modular eth roots (17 min), Introduction to Cryptography, Stanford Online
8. Boneh, Dan, Arithmetic algorithms (13 min), Introduction to Cryptography, Stanford Online
9. Boneh, Dan, Intractable problems (19 min), Introduction to Cryptography, Stanford Online
10. Shoup, Victor, A Computational Introduction to Number Theory and Algebra. Version 2, Cambridge 2008
    http://shoup.net/ntb/ntb-v2.pdf
11. http://en.wikipedia.org/wiki/Xor
12. http://www.encyclopediaofmath.org/index.php/Algebraic_independence
13. http://mathworld.wolfram.com/Function.html
14. https://en.wikipedia.org/wiki/Function_(mathematics)
15. http://www.cs.bu.edu/fac/lnd/toc/z/node24.html
16. https://en.wikipedia.org/wiki/Pseudorandomness
17. http://www.math.rutgers.edu/~erowland/modulararithmetic.html
18. http://math.wikia.com/wiki/Modular_arithmetic
19. http://www.artofproblemsolving.com/Wiki/index.php/Modular_arithmetic/Introduction
20. http://www.artofproblemsolving.com/Wiki/index.php/Modular_arithmetic/Intermediate
21. O'Neill, Adam, Stronger Security Notions for Trapdoor Functions and Applications, Atlanta 2010
    https://smartech.gatech.edu/bitstream/handle/1853/37109/oneill_adam_201012_phd.pdf
22. Boneh, Dan, Attacks on stream ciphers and the one time pad (24 min), Introduction to Cryptography, Stanford Online

23. Boneh, Dan, Real-world stream ciphers (20 min), Introduction to Cryptography, Stanford Online

24. http://www.quadibloc.com/crypto/co0408.htm

25. https://en.wikipedia.org/wiki/Stream_cipher

26. https://www.rsa.com/RSALABS/node.asp?id=2174

27. https://en.wikipedia.org/wiki/Symmetric_key_algorithm

28. http://www.ranum.com/security/computer_security/papers/otp-faq/

29. http://users.telenet.be/d.rijmenants/en/onetimepad.htm

30. http://cr.yp.to/snuffle.html

31. Bernstein, Daniel J., Salsa20 design, Chicago 2005
http://cr.yp.to/snuffle/design.pdf

32. Bernstein, Daniel J., The Salsa20 family of stream ciphers, Chicago 2007
http://cr.yp.to/snuffle/salsafamily-20071225.pdf

33. Bernstein, Daniel J., Salsa20 specification, Chicago 2005
http://cr.yp.to/snuffle/spec.pdf

34. Boneh, Dan, What are block ciphers (17 min), Introduction to Cryptography, Stanford Online

35. Boneh, Dan, The Data Encryption Standard (22 min), Introduction to Cryptography, Stanford Online

36. http://www.freesoft.org/CIE/Topics/143.htm

37. https://www.rsa.com/rsalabs/node.asp?id=2168

38. http://en.wikipedia.org/wiki/Block_cipher

39. http://www.tech-faq.com/des.html

40. http://en.wikipedia.org/wiki/Data_Encryption_Standard

41. http://www.itl.nist.gov/fipspubs/fip46-2.htm

42. https://en.wikipedia.org/wiki/Feistel_cipher

43. Boneh, Dan, Exhaustive search attacks (20 min), Introduction to Cryptography, Stanford Online

44. Boneh, Dan, More attacks on block ciphers (16 min), Introduction to Cryptography, Stanford Online

45. http://cs-exhibitions.uni-klu.ac.at/index.php?id=261

46. http://www.top500.org/lists/2012/11/

47. Boneh, Dan, The AES block cipher (14 min), Introduction to Cryptography, Stanford Online

48. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

49. http://www.cryptopp.com/wiki/Advanced_Encryption_Standard

50. https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

51. Joseph Alexander Brown, Sheridan Houghten and Beatrice Ombuki-Berman, Genetic Algorithm Cryptanalysis of a Substitution Permutation Network, Ontario http://www.uoguelph.ca/~jbrown16/NVfinal.pdf

52. https://en.wikipedia.org/wiki/Substitution-permutation_network

53. Boneh, Dan, Message Authentication Codes (16 min), Introduction to Cryptography, Stanford Online

54. Boneh, Dan, MACs Based On PRFs (10 min), Introduction to Cryptography, Stanford Online

55. Boneh, Dan, CBC-MAC and NMAC (20 min), Introduction to Cryptography, Stanford Online

56. Dworkin, Morris, Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication, Gaithersburg 2005 http://csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pdf

57. http://www.nuee.nagoya-u.ac.jp/labs/tiwata/omac/omac.html

58. Boneh, Dan, The RSA trapdoor permutation (18 min), Introduction to Cryptography, Stanford Online

59. Boneh, Dan, Is RSA a one-way function (17 min), Introduction to Cryptography, Stanford Online

60. Boneh, Dan, RSA in practice (14 min), Introduction to Cryptography, Stanford Online

61. https://www.rsa.com/RSALABS/node.asp?id=2165

62. http://technet.microsoft.com/en-us/library/aa998077(v=exchg.65).aspx

63. http://www.pgpi.org/doc/pgpintro/

64. http://www.aspencrypt.com/crypto101_public.html

65. http://www.efgh.com/software/rsa.htm

66. https://www.rsa.com/rsalabs/pkcs/files/h11300-wp-pkcs-1v2-2-rsa-cryptography-standa
pdf

Appendix

A Note on Title Formatting

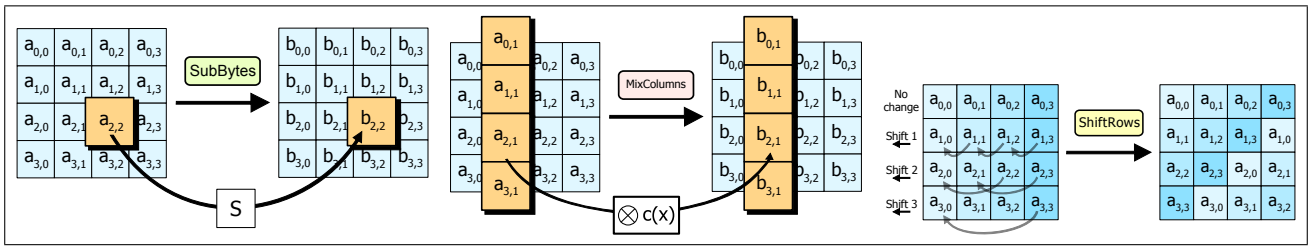
Capitalisation in titles is a widely discussed subject in English typography and there are no fixed rules for formatting titles. I did my best to emphasize the most important words in the titles in this paper and thus decided not to follow any given style.

A Note on Video Sources

The Coursera online course *Introduction to Cryptography* by Prof. Dan Boneh of the Stanford University is one of my main sources. Due to no fixed rules regarding the notation of video sources, I decided on the following scheme:
`Surname, forename, video title (video length), course name, course provider`

All videos referred to in this document are available for free, but require a user account on Coursera. Because of this I decided to include all videos referred to in the sources along with the other resources accompanying this paper.

Images



Appendix Figure 1: AES Operations SubBytes, MixColumns and ShiftRows;
https://en.wikipedia.org/wiki/File:AES-SubBytes.svg,
https://en.wikipedia.org/wiki/File:AES-MixColumns.svg,
https://en.wikipedia.org/wiki/File:AES-ShiftRows.svg

```
package main

import (
    "fmt"
    "time"
    "os"
    "path/filepath"
    "runtime"
    "hash"
    "hash/adler32"
    RNG "crypto/rand"
    Salsa20 "code.google.com/p/go.crypto/salsa20"
    DES "crypto/des"
    AES "crypto/aes"
    RSA "crypto/rsa"
)

/*
  _____       _ _    _       _    _
 |_   _|     (_) |  (_)     | |  (_)
   | |  _ __   _| |_  _  __ _| |_  _ ___  ___ _ __
   | || '_ \| | __| |/ _` | __| |/ _ \| '_ \
  _| || | | | | |_| | (_| | |_| | (_) | | | | |
  \___/_| |_|_|\__|_|\__,_|\__|_|\___/|_| |_|
*/

type MeasuredTime struct {
    label string
    time string
```

```
}

var Timers []MeasuredTime

type MillisecondsTimer struct {
    label string
    time time.Time
}

func NewMillisecondsTimer(label string) MillisecondsTimer {
    return MillisecondsTimer{label, time.Now()}
}

func (timer *MillisecondsTimer) Stop() {
    nanoseconds := time.Since(timer.time).Nanoseconds()
    milliseconds := nanoseconds / 1000000
    seconds := milliseconds / 1000

    if milliseconds == 0 {
        if nanoseconds == 0 {
            Timers = append(Timers, MeasuredTime{timer.label, fmt.Sprintf("<1ms")})
        } else {
            Timers = append(Timers, MeasuredTime{timer.label, fmt.Sprintf("%dns", nanoseconds)
                })
        }
    } else if milliseconds > 60000 {
        Timers = append(Timers, MeasuredTime{timer.label, fmt.Sprintf("%ds", seconds)})
    } else {
        Timers = append(Timers, MeasuredTime{timer.label, fmt.Sprintf("%dms", milliseconds)})
    }
}

var Measurements *os.File

func StartLogFile() {
    var err error

    cwd, err := os.Getwd()

    if err != nil {
        fmt.Println("Failed to obtain current working directory!")
        panic(err)
    }

    timeString := fmt.Sprintf("%d.%d.%d-%d_%d_%d",
        time.Now().Day(), time.Now().Month(), time.Now().Year(),
        time.Now().Hour(), time.Now().Minute(), time.Now().Second())
    Measurements, err = os.Create(filepath.Join(cwd, "cryptobench-" + timeString + ".log"))

    if err != nil {
        fmt.Println("Failed to create log file!")
        panic(err)
    }

    _, err = Measurements.Write(
        []byte(
            "CryptoBench by Jan Toennemann\n" +
            runtime.GOOS + "    " + runtime.GOARCH + "\n\n"))

    if err != nil {
        fmt.Println("Failed to write to log file!")
        panic(err)
    }
}

func EndLogFile() {
    var err error

    for _, measurement := range Timers {
        _, err = Measurements.Write([]byte(measurement.label + ":    " + measurement.time + "\n
            "))

        if err != nil {
            fmt.Println("Failed to write to log file!")
            panic(err)
        }
```

```
    }

    err = Measurements.Close()

    if err != nil {
        fmt.Println("Failed to close log file!")
        panic(err)
    }
}

/*
 _____ _        _
/  __ (_)      | |
| /  \/ _ _ __ | |__    ___ _ __ ___
| |   | | '_ \| '_ \ / _ \ '__/ __|
| \__/\ | |_) | | | |  __/ |  \__ \
 \____/_| .__/|_| |_|\___|_|  |___/
        | |
        |_|
*/

/*
    NEW CIPHER
    STREAM
    One-Time Pad
*/
func OTPEncrypt(length int) {
    OTPKey := make([]byte, length)
    OTPPlaintext := make([]byte, len(OTPKey))
    OTPCiphertext := make([]byte, len(OTPKey))

    label := fmt.Sprintf("OTP; %d kilobyte message", length / 1024)
    timer := NewMillisecondsTimer(label)
    for i, v := range OTPPlaintext {
        OTPCiphertext[i] = v ^ OTPKey[i]
    }
    timer.Stop()
}

/*
    NEW CIPHER
    STREAM
    Salsa20
*/
func Salsa20Encrypt(length int, noncesize int) {
    var Salsa20Key [32]byte
    copy(Salsa20Key[:], make([]byte, len(Salsa20Key)))

    Salsa20Nonce := make([]byte, noncesize)

    Salsa20Plaintext := make([]byte, length)
    Salsa20Ciphertext := make([]byte, len(Salsa20Plaintext))

    label := fmt.Sprintf("Salsa20: %d kilobyte message, 256 bit key, %d bit nonce", length /
        1024, noncesize * 8)
    timer := NewMillisecondsTimer(label)
    Salsa20.XORKeyStream(Salsa20Ciphertext, Salsa20Plaintext, Salsa20Nonce, &Salsa20Key)
    timer.Stop()
}

/*
    NEW CIPHER
    BLOCK
    Data Encryption Standard
*/
func DESEncrypt(length int) {
    DESKey := make([]byte, 8)

    DESPlaintext := make([]byte, 8)
    DESCiphertext := make([]byte, 8)

    DESCipher, err := DES.NewCipher(DESKey)

    if err != nil {
        fmt.Println("Failed to initiate DES cipher.")
        panic(err)
```

```
    }

    label := fmt.Sprintf("DES Encryption: %d kilobyte message, 64 bit key, %d rounds", length
        / 1024, length / 8)
    timer := NewMillisecondsTimer(label)
    for i := 0; i < length / 8; i++ {
        DESCipher.Encrypt(DESCiphertext, DESPlaintext)
    }
    timer.Stop()
}

func DESDecrypt(length int) {
    DESKey := make([]byte, 8)

    DESPlaintext := make([]byte, 8)
    DESCiphertext := make([]byte, 8)

    DESCipher, err := DES.NewCipher(DESKey)

    if err != nil {
        fmt.Println("Failed to initiate DES cipher.")
        panic(err)
    }

    label := fmt.Sprintf("DES Decryption: %d kilobyte message, 64 bit key, %d rounds", length
        / 1024, length / 8)
    timer := NewMillisecondsTimer(label)
    for i := 0; i < length / 8; i++ {
        DESCipher.Decrypt(DESPlaintext, DESCiphertext)
    }
    timer.Stop()
}

/*
    NEW CIPHER
    BLOCK
    Advanced Encryption Standard
*/
func AESEncrypt(length int, keysize int) {
    AESKey := make([]byte, keysize)

    AESPlaintext := make([]byte, 16)
    AESCiphertext := make([]byte, 16)

    AESCipher, err := AES.NewCipher(AESKey)

    if err != nil {
        fmt.Println("Failed to initiate AES cipher.")
        panic(err)
    }

    label := fmt.Sprintf("AES Encryption: %d kilobyte message, %d bit key, %d rounds", length
        / 1024, keysize * 8, length / 16)
    timer := NewMillisecondsTimer(label)
    for i := 0; i < length / 16; i++ {
        AESCipher.Encrypt(AESPlaintext, AESCiphertext)
    }
    timer.Stop()
}

func AESDecrypt(length int, keysize int) {
    AESKey := make([]byte, keysize)

    AESPlaintext := make([]byte, 16)
    AESCiphertext := make([]byte, 16)

    AESCipher, err := AES.NewCipher(AESKey)

    if err != nil {
        fmt.Println("Failed to initiate AES cipher.")
        panic(err)
    }

    label := fmt.Sprintf("AES Decryption: %d kilobyte message, %d bit key, %d rounds", length
        / 1024, keysize * 8, length / 16)
    timer := NewMillisecondsTimer(label)
```

```
    for i := 0; i < length / 16; i++ {
        AESCipher.Decrypt(AESPlaintext, AESCiphertext)
    }
    timer.Stop()
}

/*
    NEW CIPHER
    PUBLIC-KEY
    RSA
*/
var (
    RSAPlaintext []byte
    RSACiphertext []byte
    RSALabel []byte = make([]byte, 4)
    RSAHash hash.Hash32 = adler32.New()
    RSAKey *RSA.PrivateKey
    RSAKeySize int
)

func RSAGenerateKey(keysize int) {
    var err error

    RSAKeySize = keysize

    label := fmt.Sprintf("RSA key generation: %d bit key", RSAKeySize)
    timer := NewMillisecondsTimer(label)
    RSAKey, err = RSA.GenerateKey(RNG.Reader, keysize)

    if err != nil {
        fmt.Println("Failed to initiate RSA cipher.")
        panic(err)
    }

    timer.Stop()
}

func RSAEncrypt(length int) {
    var err error

    RSAPlaintext = make([]byte, length)

    label := fmt.Sprintf("RSA Encryption: %d bit message, %d bit key", length, RSAKeySize)
    timer := NewMillisecondsTimer(label)
    RSACiphertext, err = RSA.EncryptOAEP(RSAHash, RNG.Reader, &RSAKey.PublicKey, RSAPlaintext,
        RSALabel)

    if err != nil {
        fmt.Println("Failed to encrypt using RSA cipher.")
        panic(err)
    }

    timer.Stop()
}

func RSADecrypt(length int) {
    var err error

    label := fmt.Sprintf("RSA Decryption: %d bit message, %d bit key", length, RSAKeySize)
    timer := NewMillisecondsTimer(label)
    RSAPlaintext, err = RSA.DecryptOAEP(RSAHash, RNG.Reader, RSAKey, RSACiphertext, RSALabel)

    if err != nil {
        fmt.Println("Failed to decrypt using RSA cipher.")
        panic(err)
    }

    timer.Stop()
}

/*
___  ___      _
|  \/  |     (_)
| .  . | __ _ _ _ __
| |\/| |/ _` | | '_ \
| |  | | (_| | | | | |
```

```
\-|  |-/\--,-|-|-| |-|
*/
func main() {
    StartLogFile()

    fmt.Println("=============================================")
    fmt.Println("=== CRYPTOBENCH - BY JAN TOENNEMANN ===")
    fmt.Println("=============================================")
    fmt.Println("\n")

    fmt.Println("Starting tests...\n")

    fmt.Println("One-Time Pad: Encryption (1/5)")
    OTPEncrypt(1048576)
    OTPEncrypt(2097152)
    OTPEncrypt(4194304)
    OTPEncrypt(8388608)
    OTPEncrypt(16777216)

    fmt.Println("Salsa20: Encryption (2/5)")
    Salsa20Encrypt(1048576, 8)
    Salsa20Encrypt(2097152, 8)
    Salsa20Encrypt(4194304, 8)
    Salsa20Encrypt(8388608, 8)
    Salsa20Encrypt(16777216, 8)
    Salsa20Encrypt(4194304, 24)
    Salsa20Encrypt(8388608, 24)
    Salsa20Encrypt(16777216, 24)

    fmt.Println("Data Encryption Standard: Encryption (3/5) [1/2]")
    DESEncrypt(1048576)
    DESEncrypt(2097152)
    DESEncrypt(4194304)
    DESEncrypt(8388608)
    DESEncrypt(16777216)
    fmt.Println("Data Encryption Standard: Decryption (3/5) [2/2]")
    DESDecrypt(1048576)
    DESDecrypt(2097152)
    DESDecrypt(4194304)
    DESDecrypt(8388608)
    DESDecrypt(16777216)

    fmt.Println("Advanced Encryption Standard: Encryption (4/5) [1/2]")
    AESEncrypt(1048576, 16)
    AESEncrypt(2097152, 16)
    AESEncrypt(4194304, 16)
    AESEncrypt(8388608, 16)
    AESEncrypt(16777216, 16)
    AESEncrypt(4194304, 24)
    AESEncrypt(8388608, 24)
    AESEncrypt(16777216, 24)
    AESEncrypt(8388608, 32)
    AESEncrypt(16777216, 32)
    fmt.Println("Advanced Encryption Standard: Decryption (4/5) [2/2]")
    AESDecrypt(1048576, 16)
    AESDecrypt(2097152, 16)
    AESDecrypt(4194304, 16)
    AESDecrypt(8388608, 16)
    AESDecrypt(16777216, 16)
    AESDecrypt(4194304, 24)
    AESDecrypt(8388608, 24)
    AESDecrypt(16777216, 24)
    AESDecrypt(8388608, 32)
    AESDecrypt(16777216, 32)

    fmt.Println("RSA: 1024-bit key (5/5) [1/5]")
    RSAGenerateKey(1024)
    RSAEncrypt(96)
    RSADecrypt(96)
    fmt.Println("RSA: 2048-bit key (5/5) [2/5]")
    RSAGenerateKey(2048)
    RSAEncrypt(128)
    RSADecrypt(128)
    fmt.Println("RSA: 4096-bit key (5/5) [3/5]")
    RSAGenerateKey(4096)
    RSAEncrypt(192)
```

```
    RSADecrypt(192)
    fmt.Println("RSA: 8192-bit key (5/5) [4/5]")
    RSAGenerateKey(8192)
    RSAEncrypt(256)
    RSADecrypt(256)
    fmt.Println("RSA: 16384-bit key (5/5) [5/5]")
    RSAGenerateKey(16384)
    RSAEncrypt(384)
    RSADecrypt(384)

    fmt.Println("\nAll tests successfully completed.")
    fmt.Println("Please refer to the log file for detailed information.")

    EndLogFile()
}
```

Appendix Listing 1: CryptoBench.go

Erklärung zur Selbstständigkeit

"Hiermit erkläre ich, dass ich die vorliegende Facharbeit selbstständig angefertigt, keine anderen als die angegebenen Hilfsmittel benutzt und die Stellen der Facharbeit, die im Wortlaut oder im wesentlichen Inhalt aus anderen Werken entnommen wurden, mit genauer Quellenangabe kenntlich gemacht habe."

Ort, Datum und Unterschrift