University of South-Eastern Norway
Faculty of Technology, Natural Sciences and Maritime Sciences
–

Master’s Thesis
Study programme: System Engineering with Embedded Systems
Autumn 2020

Paul Knutson

Vulnerability analysis of Salsa20

Differential analysis and deep learning analysis of Salsa20.

University of South-Eastern Norway
Faculty of Technology, Natural Sciences and Maritime Sciences
Institute of Science and Industry Systems
PO Box 235
NO-3603 Kongsberg, Norway

http://www.usn.no

This thesis is worth 30 study points

Summary

This work attempts to address the research question of how secure the current solutions in lightweight cryptography are, and specifically, if Salsa20 is a sufficiently secure algorithm for its intended purposes.

We perform a state of the art survey on the current landscape of lightweight cryptography and a survey of the cryptanalysis most relevant to these kinds of crypto systems. We take a closer look at the ARX-based stream cipher Salsa20, analyse its security and give recommendation based on the results.

We implement two analyses against both Salsa20 and one of its code components, the quarter-round function. While breaking the quarter-round may not be useful for breaking Salsa20, it gives us an idea of the viability of the analysis. The two analysis methods are:

1. Differential analysis using the Hamming distance.
   We found that the quarter-round, when treated like an encryption algorithm, had an insufficient avalanche effect and is easily distinguishable from random noise for chosen plaintexts. We could not find any indication the full Salsa20 algorithm suffer from these effects.

2. Deep learning-based analysis using a context aggregation network.
   This analysis used images (some generated from random noise, some actual images), encrypted them, and tested if the context aggregation network (CAN) was able to learn and reconstruct parts of the original images or plaintexts. The results indicated this method is not viable against either Salsa20 nor its quarter-round function.

We therefore conclude that these forms of analysis does not seem effective against Salsa20.

Acknowledgments

This master thesis was written at the University of South-Eastern Norway, with the assistance of two supervisors:

1. Professor Daniel Larsson, University of South-Eastern Norway (USN), Faculty of Technology, Natural Sciences and Maritime Sciences, Kongsberg.

2. Associate Professor Kiran Raja, Norwegian University of Science and Technology (NTNU), Faculty of Information Technology and Electrical Engineering, Gjøvik.

I would like to thank my two supervisors. Their help and support have been vital to the work. Thanks to Julie for keeping me sane during the COVID-19 lockdown. Thanks to friends and family, and especially from my mother, for their support.

Contents

List of Figures

List of Tables

Abbr	Phrase	Short description
ARX	Modular **A**ddition, **R**otation and **X**OR	A type of cipher structure.
CAN	**C**ontext, **A**ggregation and **N**etwork	A machine learning/deep learning technique, which is used to generate or modify images, based on learned behaviour.
ECC	**E**lliptic **C**urve **C**yptography	A category of asymmetric cryptographic algorithms.
GE	**G**ate **E**quivalent	Estimate on how many logical gates on a processing unit an algorithm implementation requires.
HD	**H**amming **D**istance	Amount of bit-by-bit differences between two binary numbers.
HW	**H**amming **W**eight	Amount of non-zero bits in a binary number.
HW	**H**ard**w**are	Physical implementation or systems. (As opposed to virtual software systems.)
IS	**I**nternal **S**tate	The internal temporary values in a cryptographic systems.
IV	**I**nitialization **V**ector	A set of initial starting values of a cryptographic system. These are generally static values.
PRG	**P**seudo **R**andom number **G**enerator	A function or system which generates pseudorandom values, based on some seed value(s).
PRP	**P**sudo **R**andom **P**ermutation	Similar to a PRG, but makes a pseudorandom one-to-one mapping.
PT/CT	**P**lain**t**ext / **C**lain**t**ext	The input text, message, file or general data of an encryption algorithm (PT), and its encrypted version (CT).
QR	**Q**uarter-**R**ound function	The core function of the Salsa family encryption algorithms.
SPN	**S**ubstitution-**P**ermutation **N**etwork	A type of cipher structure.
SW	**S**oft**w**are	Virtual implementations or systems. (As opposed to physical hardware systems.)
XOR	E**x**clusive **or**	A function which does a bit-by-bit comparison between two binary numbers.

Table 1: List of abbreviations, with short descriptions.

Chapter 1

Introduction

As microprocessors become cheaper, the Internet of Things (IoT) become more prevalent. Because to this, we see this kind of technology take a bigger and bigger part of our lives. However, this brings up questions of computer security and privacy. We expect our data to be protected from unwanted listeners and hackers who want to steal it, modify it or destroy it. We also expect the newest technology available at low cost. As the technology for IoT is strongly connected to networks and the internet, its capability to affect our lives in negative ways grows. It is therefore important to protect our data and our devices from unwanted attackers. Cryptography is the field on how to obscure the data, so that attackers are unable to access it. This is part of what keeps the attackers out of your data, and makes sure your modern car's breaking systems hopefully cannot be disabled remotely. There exists many standardized algorithms and crypto systems today, which mostly does a seemingly good job. Crypto systems like AES and RSA are widely used and has yet to be broken. However, many of these traditional cryptography systems require a lot of computing power, energy consumption or memory. While these cryptography systems are suitable for home computers and servers, they may not be as suitable for all smaller devices. Devices like kitchen appliances, pacemakers and RFID-cards. After all, the latter devices will often have a very limited computing power, memory or energy supply. On some weaker systems, the algorithms may run slower than wanted. On others, it will not run at all. In addition to this, a lot of smaller, cheaper and weaker systems are more physically accessible to potential attackers. This can allow attackers a greater set of attack methods.

Outline of the article

In this work, we give an overview of the lightweight cryptography landscape. We focus especially on IoT looking at symmetric ciphers and hashing algorithms. We look at specific details of the algorithms, like key sizes and implementation efforts. We list the merits and demerits for various algorithms, of how suitable they are for IoT.

Of many algorithms considered for analysis, we take a deeper look at Salsa20 [1], due to its recent success as a candidate for wider use. We focus on analysing the ease of attacks to supplement the existing studies analysing the attack possibilities. In this regard, we present an overview of attacks/cryptanalysis approaches. We formulate the main research question as follows:

- Does Salsa20 seem to be secure against existing attacks?
- Can we break a weakened version of Salsa20?

To answer these questions, we look at the state of the art of Salsa20 cryptanalysis in Section 2.6. We discuss some attack methods, and how some of these can be protected against. We also analyse the Salsa20 algorithm and one of its core components, the quarter-round (QR) function. We attempt two forms of analysis, each of them against both the full Salsa20 algorithm, and the Salsa20 quarter-round function.

The first of these methods, described in Section 3.3.3, is a differential analysis using the Hamming distance (HD) for measuring distance between values. Here, we measure how the difference in multiple input values affect the difference in their output values. We do this both between an input and its output, and between two outputs with similar inputs.

The second method, described in Section 3.3.4, uses a deep learning tool called a context aggregation network, or CAN. It can be used to construct or modify images, based on data sets of input and output images. For more about CAN, see Section 2.7. Here, we encrypt plaintexts and image files. We generate images before and after the encryption, and train the network on them.

In the rest of this thesis, Chapter 2 presents the related background works, overview of the lightweight cryptography landscape relating to symmetric ciphers and hashing algorithms, overview of attack and cryptanalysis methods, how robust Salsa20 is against some of these methods and how some of the weaknesses can be reduced and protected against. In Chapter 3, we present a more detailed look into the structure of Salsa20, how its upgraded version ChaCha differs from it, and how we intend to analyse Salsa20 and its quarter-round. In Chapter 4 we discuss the results found when analysing the algorithm. And finally, in Chapter 5 we draw conclusions from our results, and we discuss further work.

Chapter 2

Background

2.1 IoT and mesh network security

Due to the reduction of costs to produce microprocessors over time, their prevalence has risen drastically over the years (see [2]). Now, everything from cars to vacuum cleaners and coffee makers can be connected to the internet. While some purposes may not require high levels of security and performance, this still remains important for other systems [3]. There has also been a higher focus on mesh networks in later years, both in your home and outside. Traditional networks has had a main access point, like the router in our homes, which connects to all the devices in your home. This is what we refer to as a star topology, and is illustrated in Figure 2.1. This topology has a few advantages over mesh networks. For example:

- Only *one* main/hub device is needed, and
- it makes the networking structurally and topologically simple.

Since you need only a single device, it makes sense to put more effort and money into making it. Therefore, you can often make it with more processing power, more memory, etc. allowing for stronger crypto algorithms with higher requirements.

On the other hand, a router's reach is limited, and connecting multiple routers together to extend the signal, is often not a practical solution. In addition, if this *one* device fails, your network is down. This is where mesh networks become useful. Instead of having a single hub all other devices connect to, like routers or cell towers, you can have multiple nodes, with multiple devices connecting to each other. In Figure 2.2, we can see an illustration of how a mesh network can look like. Mesh networks can give the network a

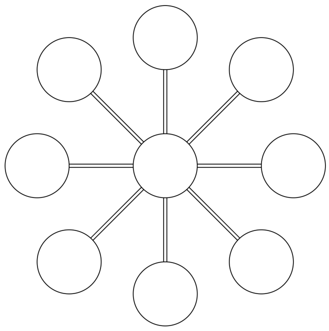

Figure 2.1: Structure of a traditional star-formed network.

robustness that star topology networks will not be able to deliver. This topology can be used for everything from small 5G cell towers around the city, to IoT devices in your home. If one node or access point fails, the network can attempt to repair itself by re-adjusting and using other nodes [4] and connections. Mesh networks can thus be less likely to fail, as often multiple nodes has to break at once. The fact that nodes can carry the signal via each other means that the network range can be extended drastically, without requiring costly long-range routers. Mesh networks can even be useful in disaster areas, where the existing infrastructure has broken down, as discussed in [4] and [5].

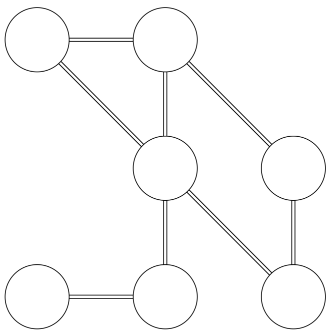

Figure 2.2: Example of structure of a mesh network.

There are some important considerations here as well:

- The nodes are often more physically accessible than a single access point would be.
- The nodes are often afforded less security-related processing power than

a single main node would.

Basically, you have to trust the nodes in the network, which makes security potentially a more relevant issue than it already is. While a large cell tower on a hill may be inaccessible to most, a small 5G node on a lamp post by your local street are easily more accessible.

Thus, to make IoT and mesh networks secure, they need to be protected against not only traditional attacks, but also physical side-channel attacks (see [6]). These are attacks not aimed only at the algorithm itself, but also the implementation of it. Using this kind of analysis, one can get access to more information about the encryption and decryption process, which can often be used to break otherwise strong algorithms. A deeper look into side-channel attack can be found in Table 2.11.

These technological advances make life more practical for millions of people, but it also has its downsides. A lack of proper cyber security can have negative and even fatal consequences. And yet many IoT products have a lax approach to security.

2.2 Basics of cryptography

In this Section, we talk about what cryptography is, what kinds of cryptography algorithms are used today, and they are used for. We talk about the two main categories of encryption algorithms; asymmetric (also known as public key) crypto and symmetric crypto, but we mainly focus on the symmetric side. We also look at hashing functions, and what they are used for.

When discussing ciphers and their strengths, words like confusion and diffusion is used a lot. It can therefore be good to understand the terms. Basically:

- **Confusion** obscures the relationship between the ciphertext and the key. Each ciphertext bit should depend on multiple parts of the key.
- **Diffusion** obscures the relationship between the ciphertext and the plaintext. If we change a bit of the plaintext, we should expect about half of the ciphertext bits to change, statistically speaking.

2.2.1 Encryption algorithms

Modern encryption can be split into symmetric and asymmetric algorithms. Symmetric algorithms uses the same key k for encryption and decryption. Asymmetric, on the other hand, uses a pair of related keys (k_1, k_2). One for encryption, the other for decryption.

Symmetric cryptography

Symmetric algorithms use the same key for encryption and decryption. This means that, when using symmetric key cryptography in communication, the key has to be known by both parties.

Symmetric ciphers can be defined by the functions (E, D), where:

- E is the encryption algorithm.
- D is the decryption algorithm.

over the sets (K, M, C), where:

- K is the set of possible keys k.
- M is the set of possible plaintext messages m.
- C is the set of possible ciphertexts c.

The encryption and decryption algorithms are defined as follows:

$$E{:}K \times M \rightarrow C$$
$$D{:}K \times C \rightarrow M$$

where the $\times$ symbol means the function E maps K and M to C (for the first line).

The defining element in symmetric algorithms, compared to asymmetric ones, is that encryption and its respective decryption uses the same key k, such that:

$$E(k, m) = c$$
$$D(k, c) = m$$

And the following should therefore be true of symmetric encryption:

$$D(k, E(k, m)) = m$$

Modern symmetric ciphers are generally split into two categories: Stream ciphers and block ciphers. The difference between these can be somewhat blurry, but generally, block ciphers encrypt the data in larger blocks, while stream ciphers encrypts 1 byte or bit at a time, usually by XORing them with the generated keystream. These are again built on different structures, as can be seen in Table 2.1. Stream ciphers often output ciphertext part by part, as the data is encrypted. They contain a hidden state which changes during the encryption, that are used for the generating the keystream (see [7]). Block ciphers requires larger chunks of the data to be encrypted before outputting any of it. One of the good things about stream ciphers is that they can output data as soon as it is encrypted. When one byte or bit is encrypted, it can be sent on to the next process. Block ciphers has to wait for the entire block to be encrypted. This means they have to, at the very least, store an entire block in memory. For this reason, block ciphers often require more memory. Stream ciphers often use simpler cipher structures and require less hardware complexity (see [8]), thus making them more suited for cheap devices with low computing power and little memory. While stream ciphers has some upsides, they also often have a lot less, if any, diffusion. Remember from earlier that the diffusion of an algorithm is how well it obfuscates the relationship between the plaintext and ciphertext. Changing a single bit in the plaintext would also change a single bit in the ciphertext, assuming the keystream is not affected by the plaintext. Thus a lack of diffusion and a bad avalanche effect for the plaintext. While this does not mean block ciphers are more secure than stream ciphers, it does mean block ciphers can have an advantage here.

Structure	Description
Iterated block ciphers	The idea of iterated block ciphers are to start with a block of plaintext of a certain size, and encrypt into a block of ciphertext of the exact same size. This is done by iterating the block through a number of rounds of encryption. These can generally be made secure by simply using a sufficient amount of rounds.
SPN	A Substitution-Permutation Network is an encryption scheme which first substitutes a specific amount of bits with another set of bits using S-boxes (look-up tables), then permutes, or shuffles, the values. This is usually done multiple times. The size of the S-boxes, the complexity of the permutation stage, the amount of rounds and the size of the full key are important factors to how secure such a structure is. For example, AES uses 8 bit S-boxes, shifts rows and columns, 10-16 rounds and 128-256 bit keys, while PRESENT, which is similar to AES, has 4 bit S-boxes, similar permutation, 31 rounds and 80-128 bit keys. The larger amount of rounds attempts to make up for the smaller S-box [2]. More about this in Table 2.4 and 2.7. On the other hand, large S-boxes can quickly become an issue when attempting to construct small implementations [9].
ARX	Modular addition, rotation and XOR. This type of structure changes the internal state first by modular addition between parts of the state. Next stage is rotation, usually through shifting bits a set amount of times. Finally, XOR between parts of the internal state. These hardware and software required for operations are relatively cheap and fast, which is part of what makes them popular (see [10] and [10]).
Feistel	Here, the plaintext is split into two halves of the same size. One of the halves are run through a round function and is XORed with the other half. This output is then stored on the other half. The two halves are then swapped, and this is repeated multiple times.
LFSR	Linear feedback shift register is a structure where the output of the next bit is affected by the previous bit. One of the good things about LFSRs are that they can be relatively small and simple. One of the big downsides of LFSRs are their linear nature, which makes them easy to attack. They should therefore be combined with non-linear elements to reduce this vulnerability.

Table 2.1: Types of cipher structures.

Asymmetric cryptography

The basic concept of asymmetric, or public key crypto, is that different, but related keys are used to encrypt and decrypt data. Using similar definitions to the symmetric encryption definition, where E is an encryption algorithm, D is its decryption algorithm, m is the message, and k_0 and k_1 are the keys, we get the following.

$$E(k_0, m) = c$$
$$D(k_1, c) = m$$

Put them together, and this should be true for asymmetric encryption.

$$D(k_1, E(k_0, m)) = m$$

In this case, k_0 would be the public key and k_1 would be the private key. Methods like Diffie-Hellman and RSA are based on the integer factoring problem. The factoring problem, relating to integer factorization, is the question of whether integer factorization can be solved in polynomial time on a classical (non-quantum) computer. This problem is considered to be hard complexity-wise. Basically, this means that it takes a lot of effort to factorize a number into its primes. Say a number a, is made by multiplying two prime numbers p and q. Finding a is easy when you know p and q:

$$p \cdot q = a$$

Similarly, finding one of the primes, when we know a and the other prime, is easy using division. For example, knowing p and a, we can find q as follows:

$$\frac{a}{p} = q$$

On the other hand, when only a is known, finding the primes is very time consuming. Many types of asymmetric cryptography systems are based on this, or other *hard* problems.

Another type of asymmetric algorithm is the elliptic curve cryptography algorithms (ECC). (Not to be confused with error-correcting code.) ECC relies on the fact that it can be very difficult to create inverse functions of certain iterated mathematical functions, over elliptic curves. They can often use short key lengths compared to other forms of asymmetric cryptography.

Many of the asymmetric algorithms are much slower or require much more computation or memory, than symmetric algorithms. For this reason, they are often just used for key exchange. Due to this, asymmetric algorithms are generally used to encrypt keys for the symmetric cryptography, which can then do the heavy lifting, so to speak. This is important for which kinds of algorithms are the most relevant for which part of networks. Especially when it comes to low-power and weak systems, which require more lightweight cryptographic algorithms.

Asymmetric algorithms may see a large shift in the future, as many of them are vulnerable to strong quantum computers (see [11] and [12]). Most asymmetric algorithms, unlike most symmetric algorithms, rely on the hard mathematical problems of integer factorization, discrete logarithm problem or elliptic-curve discrete logarithm problem. These are hard problems on classic computers, but much less so on quantum computers. While certainly interesting, post-quantum cryptography is not within the scope of this article.

Relevance to the thesis

In this study, we mainly look at symmetric key cryptography, and not asymmetric (public-key) cryptography. Part of the reason for this is that to achieve sufficient security with many popular public-key crypto, like RSA, much larger keys are required (see [13] and [14]). We considered symmetric cryptography would be the field we could contribute knowledge to the most, largely due to the author's previous knowledge of symmetric cryptography, from university and online courses.

2.2.2 Hashing algorithms

In addition to looking at lightweight block and stream ciphers, we also looked at hashing algorithms. Hashing algorithms are useful in many situations. They can be used to authenticate messages by sending along a hash of the message, showing the message has not been changed. This is similar to how MAC/MIC (Message Authentication/Integrity Code) functions work. Indeed, some MACs are built on hash functions. They can also be used as components of encryption algorithms, like Salsa20, which uses a hash function for generating its sub-keys. Hashing can be used to "store" passwords in password databases. In some ways, especially relating to passwords, hashing algorithms can be viewed as a deterministic one-way encryption. When a server checks your password, it has not actually stored your password for comparing. Not in plaintext, nor as an encrypted password. It only stores the hash of your password. That is, the output of the hashing algorithm, when your password is the input. When you log in next time, your password is hashed again, and the server can compare the hashes. Since they are deterministic, the hash will always be the same.

$$pw_a = pw_b \implies \text{hash}(pw_a) = \text{hash}(pw_b)$$

This way, servers can, in a sense, store your password without worrying as much about leaking or disclosing the passwords. Note that this is a bit of a simplification, as they should also use a salt when hashing, and maybe hash the password multiple times. Salting is done by adding another, preferably random, element into the hashing algorithms. That way, if two people have the same password (which is a very common occurrence), their hashes will be different, because their salts were different.

A hashing algorithm maps an input of arbitrary length n to an output of fixed length m:

$$\mathbb{F}_2^n \longrightarrow \mathbb{F}_2^m \tag{2.1}$$

The digest size of the hashing algorithms is then m. So whether we hash a single letter, or a massive video file, the output will always be of size m. One of the natural consequences of the change in the size from n to m is that when $n > m$, we get collisions, due to the pigeonhole principle. This principle states that when mapping n elements into m containers, where $n > m$, there will be at least one container containing multiple elements. This can of course also happen if $n \leq m$, but it is certain when this is not the case. A collision is when both a and b results in the same hash output.

$$\text{hash}(a) = \text{hash}(b) = c$$

In this case, a and b causes a collision. In the case of passwords, a login server may consider them equal. For example, if the passwords "a_strong_password" and "abcd" both result in the hash "ac6a7b8"

$$\text{hash(a\_strong\_password)} = \text{hash(abcd)} = \text{ac6a7b8},$$

the server would accept either as the correct password. Frequent collisions would thus drastically weaken passwords checking and other uses of hash functions. Guessing passwords or keys can be hard, but if there are millions of different passwords, all of which are accepted, the change of guessing one of them is drastically better. How strong a hashing algorithm is against collisions depends on how large the digest is, and how well distributed the mapping is. The digest size is the output size of the hash function, as shown in Equation 2.1. With a small digest, collisions are more likely to find. If the digest size is 2 bits, then any hash will land in one of 4 hash values, and a brute force attack should be simple. With a digest size of 1024 bits, there are 2^{1024} values to land in, and, assuming a well distributed mapping, a brute force attack would impractical. Not all hashing algorithm are as good at mapping evenly distributed hashes though. If a hypothetical algorithm only maps to half of the output values, this practically halves the digest space, and makes collisions twice as likely.

2.3 Lightweight cryptography

As processing becomes more ubiquitous, and everything is being connected to networks, the need for security increases. The rise of smaller and often weak microprocessors, is problematic when most of the cryptographic algorithms of today are focused on servers, home computers and smart phones (see [15]). Some IoT systems are too weak to run existing public-key systems like RSA, or existing symmetric cryptography standards like AES (Advances Encryption Standards). Some may be able to run them, but not at sufficient speed. This separation between traditional algorithms and lightweight algorithms has grown [3]. The attempt to find solutions that are much smaller, simpler and lighter, while being as secure, or at least sufficiently secure for their purpose, is now more important than ever. Both NIST and ISO are working on guidelines and standards on lightweight cryptography (see [15] and [16]). Some also consider the lightweight field to be too big, and propose it should be split into two sub-fields (see [2], [13] and [17]):

- Ultra-lightweight crypto
- IoT-crypto

Under this categorisation, ultra-lightweight cryptography focus on things like crypto solutions in RFID cards, pacemakers and similarly very weak systems. These would usually be hardware implementations. IoT-crypto would focus more on implementations on strong microprocessors, where there is sufficient resources to have software abstractions and memory usage. By resources, we mean the amount of computational power, memory, energy, and such. Thus the cryptography field could be split into roughly three fields, based on and sorted by levels of resources available:

- Traditional cryptography
- IoT cryptography
- Ultra-lightweight cryptography

When designing or analysing a cryptography system, one can use three general factors (see [2], [16] and [17]), and how they affect each other:

- Security
- Cost
- Performance (speed)

As illustrated in Figure 2.3, each of these factors tend to affect the others. More security, for example by having more rounds or larger key lengths, means that performance and/or cost are often negatively affected. A well-parallelized crypto system may have better performance over a serialized one, but it will also require more space and/or stronger processors. The cost factor is also strongly related to the level of lightweight of the system. If an algorithm requires a powerful and expensive processor, some companies may cheap out. Expensive systems will also not be very useful for RFID tags or extremely cheap IoT equipment. There are of course many more factors one can list, such as power and power consumption (see [2] and [13]), but this triad works as a good model.

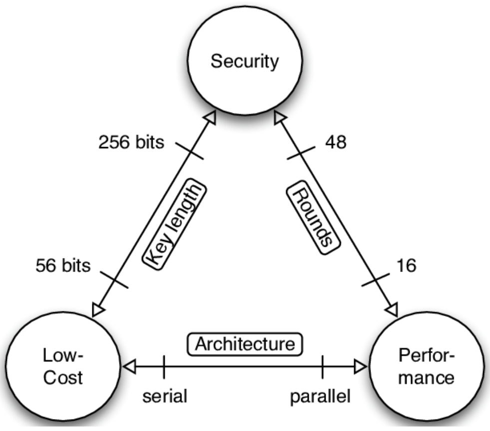

Figure 2.3: Design trade-offs in lightweight cryptography.

2.4 State of the art - Lightweight cryptography

This work is mainly based on the works of Poschmann [2], Biryukov and Perrin [3], Eisenbarth, [18] the EU ECRYPT's eSTREAM [19] and the University of Luxembourg's CryptoLUX Wiki [20]. Other references are cited in the tables.

The eSTREAM portfolio is a set of 7 ciphers, chosen in the eSTREAM project from 2004 to 2008 [19]. The algorithms, as seen in Table 2.2, are split into software and hardware algorithms.

An introductory explanation of a lot of the terms used in the tables can be found in Table 2.3. In Table 2.4, 2.5 and 2.6, we show technical information about the block ciphers, stream ciphers and hashing algorithms respectively. In Table 2.7, 2.8 and 2.9, we list up their general pros and cons, as well as known attacks (in the same order as with the technical information). Note that more known attacks does not necessarily imply weaker algorithms, or that they

Profile 1 (SW)	Profile 2 (HW)
HC-128	Grain v1
Rabbit	MICKEY 2.0
Salsa20/12	Trivium
SOSEMANUK	

Table 2.2: eSTREAM portfolio.

are considered insecure. Known possible attacks are not necessarily practical attacks. In the example of AES-256, a related-key attack using a chosen-key distinguisher by Biryukov, et al., achieves a total complexity of 2^{131} time and 2^{65} memory (see [21]). For a description on what a related-key attack is, see Table 2.10. This, while being much more effective than a brute-force attack, is still very much impractical, and AES is thus still considered secure. AES is also more popular, so there has been more research into potential attack methods than other algorithms. Similarly, a lack of pros/cons does not imply the algorithm is perfect, but rather that there are no clear, outstanding positive/negative sides with the algorithm.

Term	**Explanation**
SW and HW	SW and HW is short for software and hardware. This section is simply to tell whether the algorithm was made mainly for implementations in software, hardware or either. Of course, any algorithm *can* be made in both software and hardware, but some are more specialized for one or the other.
Key length	All symmetric ciphers use a cryptographic pseudorandom key. This is, put simply, a password used to encrypt and decrypt. How secure a key is depends on its level of entropy. Roughly speaking, how random it is and how long it is. As long as the keys are sufficiently random, the most important security characteristic of the key is its length. The longer the key, the bigger numerical space you will have to go through to guess the right one. On the other hand, longer keys means more storage required, and often more processing needed. Thus, longer keys are stronger but can also be less lightweight.
Block size	Block ciphers encrypt the data in blocks. Having large blocks may also be positive from a security standpoint. Having larger blocks also means more data will have to be stored, which is negative from a lightweight perspective.
Rounds	Most ciphers iterates a certain process multiple times. Some of the ciphers' rounds are stronger than others, but generally: More rounds means more secure, but also takes longer time.
Area (GE)	The GE, or gate equivalent, of an implemented algorithm says something about how large the algorithm has to be. Some are *much* smaller than others, which is of course positive for its lightweightness. Some have ranges, as it depends on how you implement your algorithm. For example, AES can be implemented without storing most of the S-boxes, and just generating them on the fly. This means a lot less storage, and thus a lot less GE and lower hardware requirements. On the other hand, having to generate the relevant parts of the S-box every time is slower.
Structure	As discussed earlier, ciphers can be based around different kinds of structures, with their own benefits and drawbacks. See Table 2.1 for more about this.
IV	IVs, or initialization vectors, are vectors use to initialize a state in a cipher. Some ciphers depend heavily on their internal state, and the IVs are there to ensure a proper initial state. The smaller the IV, the less memory required. On the other hand, where IVs are important, reducing their size or complexity may weaken the algorithm.
IS	The internal state is the temporary data in the algorithm, as it is processing data or keys. This has a certain size, and this size is important for the lightweightness of the algorithm. If it requires a large internal state, it will require more area, code size or memory. On the other hand, if the internal state is too small, the security of the algorithm may be compromised. For example, with a sufficiently small IS space, one may be able to guess the entire internal state. Therefore, most ciphers have an IS bigger or equal to their key size.
Digest size	A hashing function maps a variable amount of bits into a fixed size. This fixed size is the algorithm's digest.
Rate	The rate of a hashing algorithm is the size of the block being created each iteration.

Table 2.3: Explanation of terms used in the state of the art analysis.

#	Algorithm	HW/SW	Key length	Block size	Rounds	Area (GE)	Structure
1	AES	Both	128/192/256	128	10/12/14	3100	SPN
2	PRESENT	Both/HW	80/128	64	31	1075-1884	SPN
3	RC5	-	0-2040	32/64/128	12	-	ARX
4	Chaskey	SW	128	128	8/12/16	-	ARX
5	TWINE	Both	128	64	36	1800/2285	GFN
6	SPARX	-	128, 128/256	64, 128	24, 32/40	-	SPN (ARX)
7	SPECK	SW	32-128	64-256	22-34	884-1396	ARX
8	SIMON	HW	32-128	64-/256	22-34	763-1317	Feistel
9	PRINCE	-	128	64	12	3286/3491	SPN

Table 2.4: Technical information of block ciphers.

#	Algorithm	HW/SW	Key length	IV	IS	Area (GE)
1	A5/1	-	64	0	64	-
2	ChaCha20	SW	128/256	64	512	-
3	E0	-	128	0	128	-
4	F-FCSR-16/-H v3	-	128/80	128/80	256/160	-
5	Grain	HW	80/127	64	36	1294-4617
6	Mickey v2	HW	128	0-128	200/320	-
7	Salsa20	SW	256	64	256	-
8	SNOW 3G	-	128	128	576	-
9	Trivium	HW	80	80	288	-

Table 2.5: Technical information of stream ciphers.

#	Algorithm	HW/SW	Digest size	Rate	IS	Area (GE)
1	Armadillo	-	80/127/160/192/256	48/64/80/96/128	256/384/480/576/768	2923-11915
2	DM-PRESENT	-	64	80/128	64	1075-1884
3	GLUON	SW	128/160/224	8/16/32	136/176/256	-
4	Lesamnta-LW	SW	256	128	256	-
5	PHOTON	-	80/128/160/224/256	16/32/36	100/144/196/256/288	1800/2285
6	QUARK	HW	136/176/256	8/16/32	136/176/256	-
7	SipHash-2-4	-	64	64	256	884-1396
8	SPN-Hash	-	128/256	256/512	128/256	763-1317
9	Spongent	-	80/128/160/224/256	8/16	88/136/176/240/272	3286/349

Table 2.6: Technical information of hash functions.

#	Pros	Cons	Attacks
1. AES	AES is the most common and standardized symmetric cryptography algorithm, and has been for a while. Its prevalence means it has been well tested and should be relatively strong.	AES is difficult to make lightweight, and especially ultra-lightweight [22]. Among other reasons, because its large 8-bit s-box (substitution box), needing a lot of memory. It was not designed for (ultra)lightweight tasks.	• Impossible differential • Related-key boomerang • Biclique • Chosen-key distinguisher [21]
2. PRESENT	PRESENT is very similar to AES. It has a smaller s-box than AES (4-bit vs 8-bit). It is very compact, at about 2.5 times smaller than AES (see [2]). This is a big advantage for its lightweightness.	-	• Statistical saturation • Multidimensional linear • Truncated differential
3. RC5	RC5 is a simple algorithm, and has a variable block size, key size and rounds. That way it is more compatible with smaller implementations, which may need less security and less space.	When using fewer rounds (12, 64-bit blocks), it is susceptible to differential attacks using 244 specific plaintexts (see [23]). It is therefore recommended to use 18-20+ rounds. This does of course somewhat limit the advantage of simplicity.	• Differential attack • Linear attack
4. Chaskey	Chaskey is fast, secure and has a small code size [24].	It is made mainly for SW (microprocessors). Thus it is less relevant for hardware implementations. It is also a MAC, which means it's only part of a crypto system.	• Differential-linear
5. TWINE	Small and efficient, both in software and hardware [25].	-	• Biclique (full cipher) • Zero-correlation
6. SPARX	-	-	• Integral
7. SPECK	Related to Simon. Optimized for software implementations [26].	Developed by the NSA. Does not have good resistance against known-key attacks, as this was not thought of as that important for IoT crypto.	• Differential • Rectangle
8. SIMON	Related to Speck. Optimized for hardware implementations [26].	Developed by the NSA. Does not have good resistance against known-key attacks, as this was not thought of as that important for IoT crypto.	• Differential • Linear • Impossible differential • Multidimensional linear
9. PRINCE	Has a low latency compared to many other algorithms.	-	• Reflection attack • Sieve-in-the-middle • Multiple differentials

Table 2.7: Pros and cons of block ciphers.

#	Pros	Cons	Attacks
1. A5/1	Used in GSM. Wide usage often, though not always, implies good security, as they have generally been tested (and attempted broken) more.	Small key size, allows for practical attacks. A5/1 is broken.	• TMD trade-off
2. ChaCha20	Variant of Salsa20, which is part of the eSTREAM winners. Has a modified round function, which is supposed to raise the diffusion, without affecting performance.	-	• Differential attack
3. E0	Widely used as it is used in Bluetooth.	Multiple known attacks which are more efficient than brute-force, though they are seemingly not sufficiently more efficient to be practical.	• Conditional correlation
4. F-FCSR-16/-H v3	Non-linearly updated FSRs. Provably safe after a certain amount of iterations.	Allows for a significant amount of collisions during the first clockings, before the main cycle is reached.	-
5. Grain	Part of the eSTREAM winners. Based on two different FSRs, where their clocking affects each other to achieve non-linearity.	-	• Linear approximations • Dynamic cube testers
6. Mickey v2	Part of the eSTREAM winners.	-	• Entropy loss • Weak keys
7. Salsa20	Part of the eSTREAM winners [19]. No published attacks as of 2015. Secure against timing side-channel attacks.	Weak against some types of side-channel attacks (see [27] and [28]). Lacks research into its security (see [29]).	-
8. SNOW 3G	Chosen by the 3GPP consortium. [30]	Uses the (large) AES S-box.	• Multiset distinguisher
9. Trivium	About: Block-stream-cipher. XORs with bits from an external source to reduce the need for a large S-box. Uses 3 internal LFSRs. There is also "Bivium", which uses only 2. Trivium saves space due to the lack of a large S-box.	The external bits could be biased, and thus weaken the security of the algorithm.	• Algebraic (Bivium) • Conditional differential

Table 2.8: Pros and cons of stream ciphers.

#	Pros	Cons	Attacks
1. Armadillo	-	It is large and it has been broken.	• Local-linearization (practical)
2. DM-PRESENT	Based on PRESENT, which is lightweight and fairly secure.	The small digest size makes it less collision resistant.	• Multi-differential (collisions: 12 rounds, distinguisher: 18 rounds)
3. GLUON	Based on FSCRs, which can be both good and bad.	-	• Iterated preimage attack
4. Lesamnta-LW	Reuses AES-elements, making it more secure.	Reuses AES-elements, making it larger and less lightweight.	• Integral
5. PHOTON	Inspired and based on PRESENT and AES, which seems like a pretty LW and secure algorithm. Reuses from AES or PRESENT based on need.	-	-
6. QUARK	Hardware oriented and fast.	-	-
7. SipHash-2-4	-	Not collision resistant (due to small digest size). Large.	-
8. SPN-Hash	Probable security against differential collision attacks. Based on an AES-like SPN.	Similar issues as other AES-based systems. Large.	-
9. Spongent	Based on a modified version of PRESENT. No known successful attacks on Spongent.	-	• Linear distinguishers

Table 2.9: Pros and cons of hashing algorithms.

2.5 Cryptanalysis

Through the years, many types of attacks has been used and proposed against different kinds of cryptographic algorithms. This section will go through some of these, with the main focus being on symmetric ciphers.

Firstly, what kind of attack can be used depends on how much information the attacker has access to. For a list of types, see Table 2.10.

Knowledge	**Description**
Cipher-text only	Here, the attacker only has access to the cipher-text. The attacker knows nothing about the key, the plain-text/data, etc.
Chosen plain-text or cipher-text	In this case, the attacker can chose the plain-text or cipher-text. This may allow them to change it to ascertain more information about, for example, the key.
Known plain-text	In this case, the attacker knows the plain-text, but has not, and can not, choose the plain-text themselves.
Adaptive chosen plain-texts	Here, the attacker decides the plain-text and can change it as they will. The attacker can for example look for differences in cipher-text based on differences in plain-text.
Related-key attack	Sometimes keys are related, and here, the attacker can know the relation, without necessarily knowing the keys.

Table 2.10: Amount of knowledge the attacker has access to.

Usually, the attacker wants to find the key or the plaintext. Of course, finding one can often make finding the other rather simple. In the cryptography community, it is well accepted that one should always expect the attacker to know everything about the algorithm. Hiding the algorithm, while maybe making it somewhat more difficult to crack, is usually not a sufficient defense against attacks. Many older algorithms, made as in-house crypto solutions, have later been analysed, leaked or disclosed, decompiled, etc, and subsequently cracked (see [3]). This is related to whether algorithms and their cryptanalysis should be public and open-source. When algorithms are closed-source, it may be better hidden from attackers, and thus harder for attackers to analyse. On the other hand, they are therefore often analysed less by security experts, and it is less likely that potential security holes are found. This is why open-source crypto algorithms are the expected norm these days.

To break algorithms, you often need a large amount of one or more of the following:

- Time
- Data
- Memory

In a simple brute-force attack, where all key combinations are checked, you will need a lot of time. This is usually on "the age of the universe" time scales, as the key search space is vast. On the other hand, if some form of adaptive chosen plain-text attack is attempted, a huge amount of data may be needed. Similarly, sometimes it is needed to store large amounts of data.

Sometimes, one can achieve only a partial attack, or an attack where some knowledge is found, but not all. As an example, modern algorithms attempt to make the ciphertext indiscernible from random noise. If one can implement an attack or analysis which is able to distinguish between random noise and the ciphertext from an algorithm, one may be able to use that in a more complete break. This is known as a distinguishing attack. In other cases, one may be able to retrieve information about a hidden closed-source algorithm, and use it to rebuild a functionally equivalent algorithm.

In Table 2.11 we can see a list of many types of attacks. Some more general than others. This is not a comprehensive list, but gives an insight to the cryptanalysis landscape for symmetric ciphers.

Attack type	Description
Brute-force attack	This is the simplest type of attack. Essentially try all of the combinations. This is generally the worst case scenario in a break. This is what you do with a bike lock you have forgotten the key/password for. Works great for bike locks, phone locks (without hardware-blocking for too many attempts) and bad (short) passwords. When using 128 bit keys, on the other hand, it takes too long to be practical.
Differential analysis	This kind of attack analyses how changes in the input affects the output. What happens with the output of the algorithm when a single bit is changed in the key? If one can find patterns in the differences here, one may be able to recover the original key, for example by using hill climbing algorithms.
Distinguishing attack	This kind of attack or analysis attempts to find separate the ciphertext from random noise, and uses that to attack the algorithm.
Boomerang attack	The boomerang attack is a type of differential attack primarily used for breaking block ciphers. This enables us to attack specific parts of the block cipher.
Linear cryptanalysis	Mostly used for block ciphers. This method attempts to build linear equations relating key to plain-text and cipher-text and find biases.
Integral cryptanalysis	This is mostly relevant against SPN networks. Using multiple sets of pairs of related chosen plain-texts, where only parts of the plain-texts are differing from each other, while most of the bits remains the same. This is also knows as the square attack, and is similar to the saturation attack.
Meet-in-the-middle attack	This method works when multiple operations in the encryption scheme is performed in sequence, and "meeting in the middle" of the set of operations. This makes it such that it will take *more* storage, but *less* time to break the algorithm.
Related-key attack	This is, as touched on previously, a kind of attack where the attacker can use known relations between multiple keys to gain more information. A simple example of this could be that the first n bits of the keys are always the same or if someone uses keys which simply counts up 0, 1, 2, 3, etc.
Side-channel attacks	A side-channel attack is a type of attack where one does not attack the algorithms logic, but attacks the implementation. This kind of attack has been more relevant lately, especially relating to the rise of IoT and SaaS. More direct and fast communication with servers can enable attacks to gain more information about timings and such, and IoT devices are often more easily available to potential attackers. Even highly regarded algorithms can be vulnerable to many of these attacks, as they have not had the same focus for as long.
Timing analysis	This kind of side-channel attacks assume the server, memory, microprocessor, etc takes different amounts of time based on what it is doing, which enables attackers to gain important information. For a simple example: If I claim that $9257 \times 1248 = 11586166$, you may have to use a calculator to check it, and it will thus take time. If I claim it is equal to 6, you can reply immediately. If a server does this, it can tell me if your guess is close or far off.
Power analysis	If the attacker has access to the power supply or to measuring devices which can electromagnetic radiation in the microprocessor, they may be able to read out what the algorithm is doing. If, say, the attacker knows AES is being used, and sees 16 spikes in the power usage, there is a good chance AES with 16 rounds is being used. If this is not masked for, the power analysis is accurate enough and the attacker has a thorough understanding of the algorithm, they may be able to read key-bits out from the power graph.

Table 2.11: Cryptanalysis techniques and attacks for symmetric ciphers.

2.6 Cryptanalysis of Salsa20

We looked deeper into the security of Salsa20, and how much research there is on attack methods. This work focuses on the Salsa family, partly because Salsa algorithms has been adopted in multiple important standards and software solutions, and partly because some studies indicate a lack of security research (see [31]). The discussed analysis and attack methods can be found in Table 2.12. We found that the most realistic attacks seems to be the ones where we assume knowledge from side-channel analysis, to perform cryptanalysis on Salsa20.

We also looked a bit at a few relevant methods to protect against side-channel attacks, which can be seen in Table 2.13.

2.7 Context aggregation networks

We used a context aggregation network, or CAN, for parts of our research. This section talks a bit about what that is.

Deep learning is a form of machine learning based on artificial neural networks (ANN), using multiple layers for extracting features from raw input data. These types of networks are used in fields like social networks, computer vision, natural language processing, speech recognition, biometrics, translation, and even winning quizes, like when IBM's Watson won the American quiz show Jeopardy (see [34]).

A context aggregation network is a form of deep learning network, built and based upon convolutional neural networks (CNNs). It was created in 2019 by W. Cheng. et al. for semantic labeling in aerial images (see [35] and [36]), and was applied in various other applications. We were unable to find any previous research into these networks being used for cryptanalysis.

Attack type	Attack type description	Salsa20 vulnerability
Non-side-channel attacks	Create a function, or set of functions, which always, often or sometimes give you knowledge about the key, regardless of implementation method.	Salsa20 seems to be relatively strong against these kinds of attacks, though there is a lack of research here. ChaCha seems even stronger, due to higher diffusion in its round-function.
Timing attacks	Measure how long the cache, server, device etc. uses to reply. If the algorithm takes different amounts of time based on the key (like size of the key, bits of the key, errors in the key, etc.), you can learn something about the key, and thus be able to reduce the search space.	Not affected. Salsa20 is constant-time independent of input, and thus gives the attacker no information.
Power attacks	Measure the power usage, or the EM radiation, of the circuit. By knowing the setup of the algorithm, you can find out where, for example, bits are added together, multiplied etc. Worst case scenario is that you can read out the key bits almost directly, by measuring when it performs a multiplication carry or not.	Salsa20 is relatively weak against these kinds of attacks, due to the kinds of operations it uses. (Word-addition, etc).
Bricklayer attack	Optimized attack based on a divide-and-conquer strategy (see [28]).	One of the attacks we looked at. This attack seems promising against weakened versions of Salsa20, like Salsa20/8.

Table 2.12: Potential attacks against Salsa20.

Method	Description	Effective against	Pros and cons
Masking	Obscure internal variables by splitting sensitive temp variables into a set amount (masking order + 1) of parts. This requires generating pseudo-random numbers.	Most kinds of attacks [32].	+ Secure against most attacks [33]. − Often very expensive, and thus not an optimal solution for lightweight systems [33].
Code polymorphism	Hide code functionality from being read, by constantly changing the code [32]. Generating code variants statically (multi-versioning) or at runtime (generates different code efficiently and periodically). Uses a bunch of different configurations to change the code here and there, making too many possibilities to brute force. Tools like Odo can be used in this regard [32].	**Runtime** (most attacks, specified): • Register shuffling • Instruction shuffling • Semantic variants • Insertion of noise instructions	**Statically**: Limited by the final size of the program, as generating more variants induces an increase of code size. **Runtime**: Runtime code generation is usually avoided in embedded systems due to the potential vulnerability of accessing memory with both write and execution permissions.

Table 2.13: Some existing protection methods against cryptographic attacks.

Chapter 3

Methodology

The Salsa family consists of a set of stream ciphers. They were created by Daniel J. Bernstein, and they are widely used. They can be found in the eSTREAM portfolio [19] and in the TLS cipher suites [37].

In this section, we look at the construction of Salsa20 and how its updated version ChaCha differs from the original. Salsa20 is built on an ARX structure, and so first, we discuss what that is. We then run through a detailed description of how Salsa20 is constructed, and of how ChaChas design differs. Finally, we look at our methods of analysis.

3.1 Detailed description of Salsa20

3.1.1 ARX

Salsa20 is built using an ARX structure in its core. ARX is an abbreviation for modular **A**ddition, **R**otation, **X**OR (exclusive or). This section explains what each of these elements does.

Modular addition

Modular addition addition is performed by adding two numbers together, and then reducing the result with a given (fixed) modulus. We denote modular addition with the symbol $\boxplus$. In the case of Salsa20, two 32 bit numbers are added into a third number. If this number requires more than 32 bits, we simply cut off the leading bits, and make it into a 32 bit number. In Equation 3.1, we can see an example of normal addition and modular addition:

$$\begin{aligned} 1001 + 1001 &= 10010 \\ 1001 \boxplus 1001 &= 0010 \end{aligned} \tag{3.1}$$

In line 1, we add two 4 bit numbers, resulting in a 5 bit number. In line 2, we add the same two 4 bit numbers, but reduce the result with modulus 4. This results in a 4 bit number.

Rotation/shift

The rotation, or shift, simply shifts the bits a set amount. In the case of the Salsa family, the shift is to the left. A shift of n bits is denoted as $\lll n$. For example:

- abcdefg $\lll 3 =$ defgabc
- $00000001 \lll 7 = 10000000$

XOR

Finally, XOR, or exclusive or, is a Boolean operation, working as a bit-by-bit difference comparison. We use the symbol "$\oplus$" to denote an XOR operation. If the two compared bits are different, the resulting bit is a 1. Otherwise, it is a 0. As an example:

$$\begin{aligned} &00110001 \\ \oplus\ &10001100 \\ =\ &10111101 \end{aligned}$$

3.1.2 Overview of Salsa20

Salsa20, like many other stream ciphers, is XORing the plain-text data with a sub-key keystream. This means the size of the keystream has to be the same as the size of the data or data block. This is much the same concept as a one-time pad (OTP). While OTPs are unbreakable, they are rarely used, as key exchange would be impractical (see [38]). Therefore, to keep the key at a reasonable size, Salsa20 has a pseudorandom generator (PRG), which uses an expansion function and hash function, to generate the keystream. This is illustrated in Figure 3.1. The task of this function is to have the key as well as the nonce, block number and IV, as input. It then expands the key to many, large sub-keys. Specifically, expand a 16 or 32 byte key into as many 64 byte sub-keys as needed to encrypt the data. This way, you will not have to send gigabytes worth of keys to be able to send large files. You will only have to send a small key, and Salsa20's expansion function will expand it. Since this function is expanding a seed into a set of pseudorandom numbers, it is essentially a pseudorandom generator (or PRG).

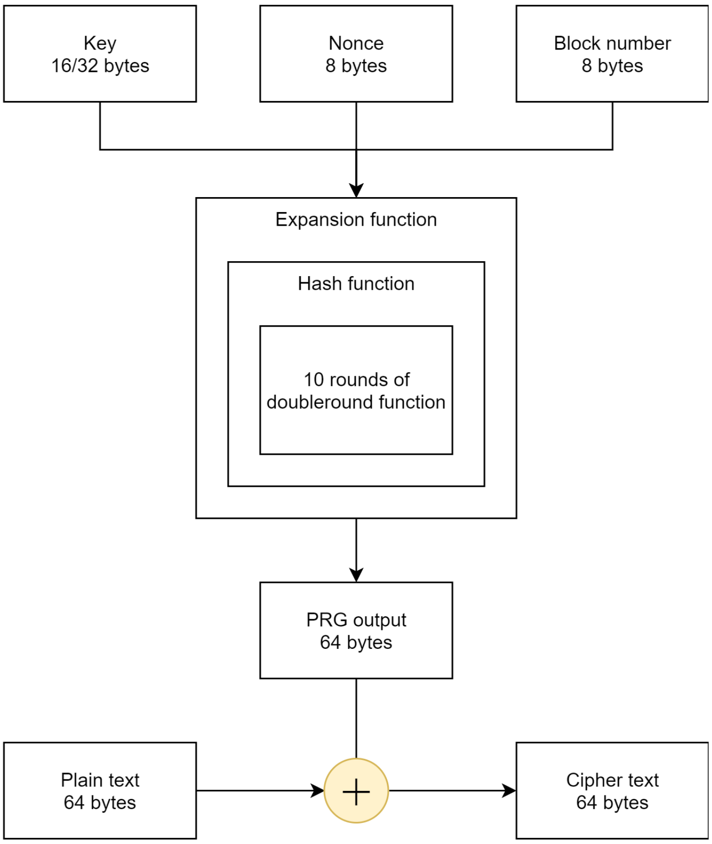

Figure 3.1: Overview of the structure of Salsa20.

Once a sub-key is created, it can be XORed bit-by-bit with 64 bytes of the unencrypted plain-text data. In the case of decryption, the cipher-text is XORed and you get back the plain-text. This can also be seen in Figure 3.1.

The PRGs 64 byte (512 bit) input consists of the following:

1. Key (32 bytes).
2. IVs (16 bytes).
3. Cryptographic nonce (8 bytes).
4. Block/counter number (8 bytes).

The key, nonce (which we will define soon), block number and the internal initialization vectors (IV) are combined to make the internal state (IS). Salsa20 has two set of IVs; one for each key-size mode. These IV sets are defined as shown in Table 3.1 (see [1]).

A	**B**
0. $(101, 120, 112, 97)$	0. $(101, 120, 112, 97)$
1. $(110, 100, 32, \mathbf{51})$	1. $(110, 100, 32, \mathbf{49})$
2. $(\mathbf{50}, 45, 98, 121)$	2. $(\mathbf{54}, 45, 98, 121)$
3. $(116, 101, 32, 107)$	3. $(116, 101, 32, 107)$

Table 3.1: Salsa20's initialization vectors (IVs).

The differences between A and B are highlighted with bold text. Running these numbers through an ASCII conversion table, we get the IVs as "expand 32-byte k" for IV A (32 byte key) and "expand 16-byte k" (16 byte key) for IV B. The initial state (IS) of Salsa20 can be arranged as a 4×4 matrix. Table 3.2 shows the matrix for a 32 byte key (IV A), where the key cells contains 4 bytes of the key, the nonce cells contains 4 bytes of the cryptographic nonce, and the block cells contains 4 bytes of the block number.

"expa"	key	key	key
key	"nd 3"	nonce	nonce
block	block	"2-by"	key
key	key	key	"te k"

Table 3.2: Salsa20's initial state (IS) for 32 byte keys.

Salsa20 supports both 32 byte keys and 16 byte keys. These are managed as follows:

- When using a 32 byte key $\rightarrow$ Split the key k into two sub-keys k_0 and k_1. Use IV A.
- When using a 16 byte key $\rightarrow$ Apply the key k twice. Use IV B.

We can see how these different parameters are given to the hash function, in Equation 3.2. In this case, k (line 2) is a 16 byte key, while k_0 and k_1 (line 1) are the two parts/sub-keys of a 32 byte key. Thus, the expansion function receives the key or key bits, as well as the nonce. It calls the hash function with the split keys (k_0 and k_1) or duplicated keys (k), the nonce (n) and the IVs as input parameters, totalling 64 bytes. The hash function returns 64 bytes of pseudorandom data. This data is then XORed with the initial input, and becomes the first sub-key. For more sub-keys, which are required when the data is bigger than 64 bytes, repeat the same process, but after incrementing the block number.

$$\begin{aligned} \mathrm{Exp}_{32}(k_1, k_2, n) &= \mathrm{Hash}(a_0, k_0, a_1, n, a_2, k_1, a_3) \\ \mathrm{Exp}_{16}(k, n) &= \mathrm{Hash}(b_0, k, b_1, n, b_2, k, b_3) \end{aligned} \tag{3.2}$$

Here, Exp refers to the expansion function, Hash refers to the hash function, and $_{32}$ and $_{16}$ refers to the size of the key in bytes.

A cryptographic nonce is usually a random number, which is used for a single encryption/decryption pair. This is to add even more randomness to the algorithm. It can also be based on hardware states, as is done in the stream cipher Trivium [39]. Ensuring the nonce is not reused is hard in lightweight crypto, as it either has to use non-volatile flash memory or base it on the hardware [40]. The block number is a binary number counting up for every block of data being processed in an encryption or decryption. So the first block of data will have block number $00\ldots00$, the second will have block number $00\ldots01$, etc. This ensures the initial state per block, is never the same, as long as the nonce is changed at least every 2^{64} blocks. Otherwise, the block number will overflow and return to $00\ldots00$, and this initial state will be used twice. Using the same hash input twice may allow an adversary to attack the algorithm.

The PRG, seen in Figure 3.1, consists of an expansion function, which then uses a hash function, which again uses the double-round function. The double-round function consists of 1 full round of the row-round function, and one round of the column-round function. Each of these functions again consists of 4 rounds of the quarter-round function, applied differently to the different parts of the IS. The quarter-round function (QR for short) is performed a total of 80 times in Salsa20, and a reduced amount in Salsa20/12, Salsa20/8, etc. The /8 means a reduced form of Salsa20, with only 8 full rounds, rather than 20.

The hash function receives a 64 byte input $(b_0, b_1, \ldots, b_{63})$. It uses these to construct 16 words $(w_0, w_1, \ldots, w_{15})$, each of 4 bytes. These are created using the Littleendian (Lit) function, which will be defined soon, as described in Equation 3.3.

$$\begin{aligned} w_0 &= \mathrm{Lit}(b_0, b_1, b_2, b_3) \\ w_1 &= \mathrm{Lit}(b_4, b_5, b_6, b_7) \\ &\dots \\ w_{15} &= \mathrm{Lit}(b_{60}, b_{61}, b_{62}, b_{63}) \end{aligned} \tag{3.3}$$

It then runs these words through the double-round (referred to as Dr) function 10 times:

$$(z_0, z_1, \dots, z_15) = \mathrm{Dr}^{10}(w_0, w_1, \dots, w_{15})$$

Finally, the hash function combines all these by concatenating a series of Lit functions, as shown in Figure 3.4.

$$\begin{aligned} &\mathrm{Lit}(z_0 + w_0) \\ &\mathrm{Lit}(z_1 + w_1) \\ &\quad\dots \\ &\mathrm{Lit}(z_{15} + w_15) \end{aligned} \tag{3.4}$$

The Lit function received 4 bytes, and changes their order. It does so as shown below:

$$\mathrm{Lit}(b) = b_0 + 2^8 b_1 + 2^{16} b_2 + 2^{24} b_3$$

Note that the + symbols in this case implies concatenation, not addition or modular addition.

The double-round function consists of a row-round function and a column-round function, both of which are applied once during a double-round, as shown below:

$$\mathrm{Dr}(x) = \mathrm{Rr}(\mathrm{Cr}(x))$$

In this equation, Dr refers to the double-round function, Rr refers to the row-round function, Cr refers to the column-round function and x is a 16 word input. The row-round and column-round both consist of 4 quarter-rounds. When placing the 16 words into a 4×4 matrix, as shown in Table 3.3, the row-round function takes row by row, while the column-round function takes column by column.

In the illustration in Figure 3.2 we can see the ARX structure elements in the quarter-round. It consists of a set of modular additions, bit-shift rotation and XOR functions. The quarter-round function is invertible, as all of its operations are invertible. The Salsa20 QR function takes a 16 byte/128 bit X as an input and outputs a 16 byte/128 bit Y. A full round performs 4 quarter-rounds, making sure all parts of the internal state is run through the

x_0	x_1	x_2	x_3
x_4	x_5	x_6	x_7
x_8	x_9	x_{10}	x_{11}
x_{12}	x_{13}	x_{14}	x_{15}

Table 3.3: The 16 binary words being used in the double-round function.

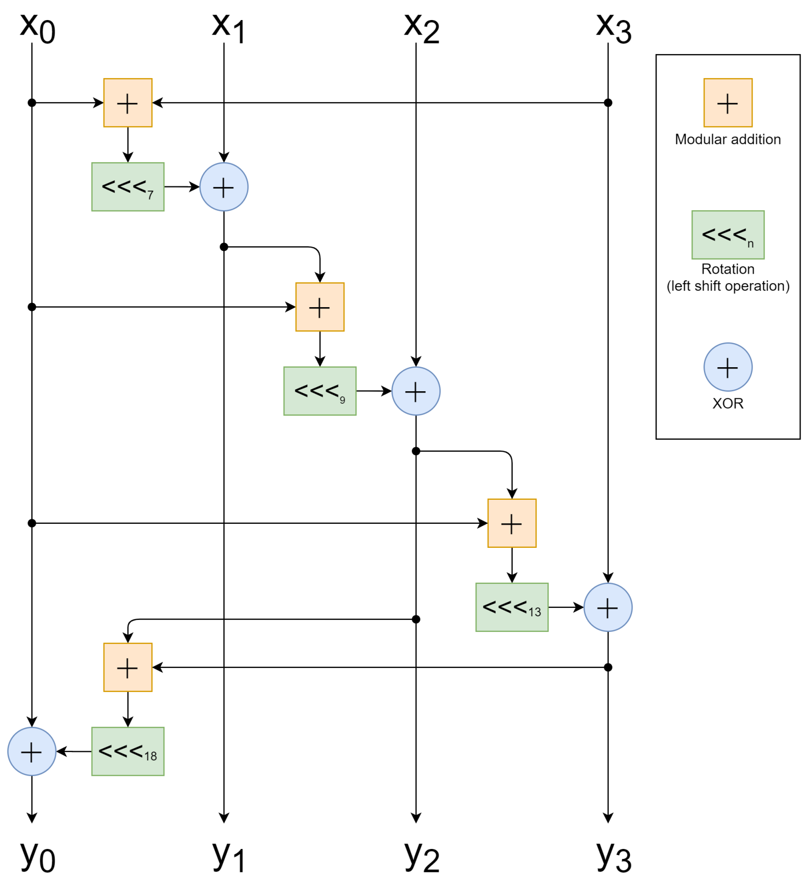

Figure 3.2: Flowchart of the Salsa20 quarter-round function.

QR function every round. The X and Y are split up in 4 pieces, each 4 bytes/32 bits, as shown in Equation 3.5.

$$
\begin{aligned}
&X = [x_0, x_1, x_2, x_3] \\
&Y = [y_0, y_1, y_2, y_3] \\
&x_i, y_i \in \mathbb{F}_2^{32}
\end{aligned}
\tag{3.5}
$$

The QR function can be written out as shown in Equation 3.6.

$$
\begin{aligned}
y_1 &= x_1 \oplus ((x_0 + x_3) \lll 7) \\
y_2 &= x_2 \oplus ((y_1 + x_0) \lll 9) \\
y_3 &= x_3 \oplus ((y_2 + y_1) \lll 13) \\
y_0 &= x_0 \oplus ((y_3 + y_2) \lll 18)
\end{aligned}
\tag{3.6}
$$

3.2 How ChaCha differs from Salsa20

ChaCha offers a solution to one of the potential issues in Salsa20's quarter-round function. In the Salsa20 QR, y_1 is affected by x_0, x_1 and x_3, but not x_2. On the other hand, y_0, y_2 and y_3 is affected by all of X. This means y_1 has less diffusion, and is thus weaker against potential attacks. ChaCha has updated this by using a modified QR function. It should diffuse more, as all output words are affected by all input words (see [41] and [42]). This is done without making it slower. In fact, in some cases it can improve speed and use less temporary memory (see [41]).

When describing the ChaCha quarter-round, we use slightly different notation than with Salsa20. Rather than (x_0, x_1, x_2, x_3) as inputs and (y_0, y_1, y_2, y_3) as outputs, we use (a, b, c, d) for inputs, and we update them directly. This means implementations of ChaCha can save some memory, where Salsa20 would require storing temporary variables.

The ChaCha quarter-round is described in Equation 3.7. The operations are performed left to right and top to bottom, similar to how it would be programmed in C. An illustration of the ChaCha quarter-round function is shown in Figure 3.3.

$$
\begin{aligned}
&a \mathrel{+}= d, \quad d \mathrel{\oplus}= a, \quad d \lll 16 \\
&c \mathrel{+}= b, \quad b \mathrel{\oplus}= c, \quad b \lll 12 \\
&a \mathrel{+}= d, \quad d \mathrel{\oplus}= a, \quad d \lll 8 \\
&c \mathrel{+}= b, \quad b \mathrel{\oplus}= c, \quad b \lll 7
\end{aligned}
\tag{3.7}
$$

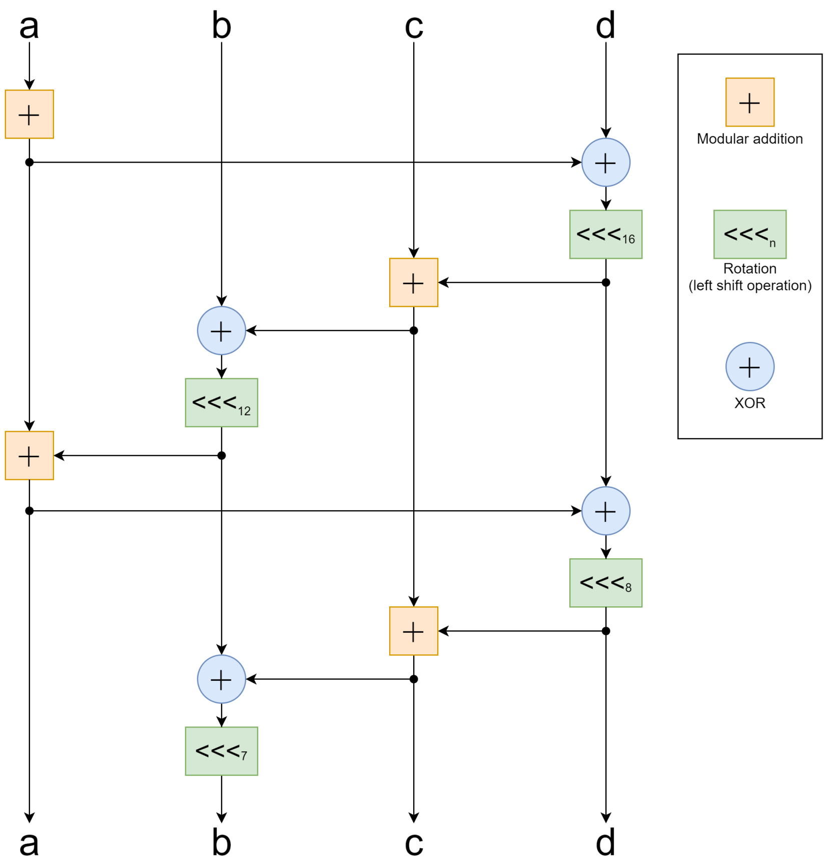

Figure 3.3: Flowchart of the ChaCha quarter-round function.

3.3 Analysis on Salsa20

In this section, we describe our how we analyzed Salsa20 and its quarter-round function. We perform all analyses on both a full Salsa20 encryption and the quarter-round function. While breaking the quarter-round function is not very useful in and of it self, it is interesting to compare it with Salsa20. Our analyses should presumably perform better against the quarter-round function, than against Salsa20. We analyse 3 methods in this section:

- Brute-force attacks
- Hamming distance-based analysis
- CAN-based analysis

3.3.1 Test setup

First, we look at the brute-force attack. While a brute-force attack is guaranteed to be successful, it is also very ineffective and slow. We therefore estimate our implementation against a brute-force attack.

We implemented a test setup of Salsa20, which we used in this analysis (see [43]). It is built in Python 3.7, and was run on an Intel i7-4790k processor running Windows 10 (x64).

The CAN-based analyses were implemented in Matlab 2020b, based on a modified image processing tool from MathWorks. The same test setup from earlier, trained the models on a NVIDIA GTX 980 Ti. The rest was implemented in Python 3.7.

3.3.2 Brute-force estimation

About brute-force attacks

The basics of a brute-force attack is to guess all of the possible values, until one works. Say the deterministic function f takes input A and returns and output B. We write this as $f(A) = B$. We guess or count new A' for all possible As, where $f(A') = B'$. Whenever the guessed and real output are the same, we can conclude either the inputs are the same, or that we have found a collision.

$$B = B' \quad \Longrightarrow \quad f(A) = f(A') \quad \Longrightarrow \quad A = A'$$

Estimating how long a brute-force attack would take on our setup

We run tests on the Salsa20 QR function and the ChaCha QR function. We also run tests on the Salsa20 full encryption and decryption, using 32 byte (256 bit) keys. All of these tests were performed with random values. The results can be found in Table 3.4, where rate $= \frac{\text{runs}}{\text{time}}$.

Function	**Runs**	**Time** (s)	**Rate** (1/s)
Salsa20 QR	100000	10.472...	9548.432...
ChaCha QR	100000	9.784...	10219.841...
Salsa20 encrypt	100	7.261...	13.771...
Salsa20 decrypt	100	7.155...	13.974...

Table 3.4: Results from the speed performance test.

We can see how the Salsa20's and ChaCha's QR function run at roughly the same rate (10000 per second). Salsa20 encryption and decryption also run at roughly the same rate (14 per second).

Since the QR function takes a 128 bit binary number, the total possible combinations of inputs are 2^{128}. A complete search would thus take about:

$$\frac{2^{128}\ QR}{10000\ \frac{QR}{s}} = 3.4 \cdot 10^{34}\ s,$$

or about 10^{27} years. Assuming we can know the cryptographic nonce used, a complete search of the 256 bit key space would take roughly:

$$\frac{2^{256}\ \text{runs}}{14\ \frac{\text{runs}}{s}} = 2.4 \cdot 10^{76}\ s,$$

or about 10^{68} years. While one could expect to find the correct Salsa20 key or QR input in about half of the time for a complete search, either are still very much impractical. We can thus conclude that a brute force attack on either the key or the QR input on either our Salsa20 or our ChaCha implementations is infeasible. While there are much faster and effective implementations of the Salsa algorithms, 128 bit keys and 256 bit keys are generally considered practically secure.

3.3.3 Hamming distance differential analysis

In this section, we look at a Hamming distance-based form of differential cryptanalysis.

Hamming weight and Hamming distance

The Hamming weight (HW) of a binary number is the amount of non-zero bits. So $HW(00100100) = 2$, as the number has two 1's. The Hamming distance is the amount of bit-by-bit difference between two binary numbers. So $HD(1000, 0000) = 1$, as the first bit is different between the two numbers. This can be formulated into the following pseudocode, where Q and P are the two numbers, and bit_q and bit_p are the bits currently being compared.

```
HD = 0
for bit_q, bit_p in Q, P:
    if bit_q =/= bit_p:
        increment HD
```

Since XOR also does a bit-by-bit difference check, we can alternatively write the HD function as:

$$HD(1000, 0100) = HW(1000 \oplus 0100) = HW(1100) = 2.$$

Differential analysis

We take a random input X and output Y, such that $f(X) \to Y$, where f is the encryption function we attempt to analyse. We then modify X into X', by flipping n bits. Run it through the encryption function f: $f(X') \to Y'$. The question now is, after n flipped bits between X and X', how many bits are flipped between Y and Y'? Or more specifically, if

$$HD(X, X') = n \to HD(Y, Y') = m,$$

what is m? Does a static n mean m is also static? If not, what is its average value for different values of n? See Figure 3.4 for an overview of the analysis method. If we can use this to decide whether or not a random X' is close to

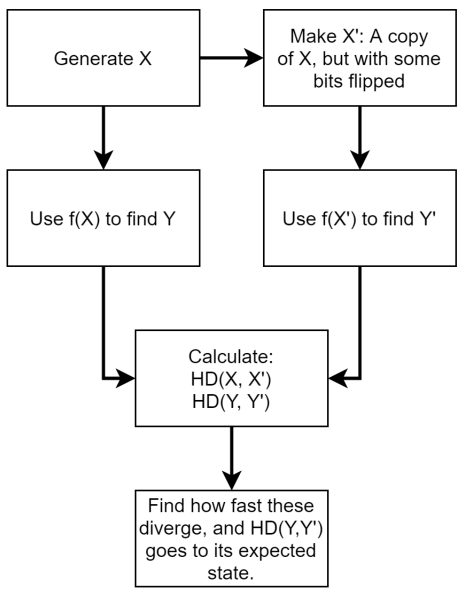

Figure 3.4: Algorithm for the differential HD analysis on X-Y pairs.

X, by looking at the Y' and Y, we may be able to use this in an attack against the algorithm. We perform this analysis on both the Salsa20 quarter-round function, and on Salsa20's PRG.

If the algorithm has a good avalanche effect, you would expect the HD to be about the same as the HD between two random values: About half the bits. More formalised, if Q and P are two random binary numbers of length n, we would expect:

$$HD(Q, P) \approx \frac{n}{2}$$

This is our null hypothesis. If, however, what we see differs from this expected value, it illustrates a lack of proper avalanche effect of the function.

3.3.4 CAN analysis

In this section, we describe how we perform a known-ciphertext analysis using a context aggregation network, or CAN. Since these networks are made for creating or modifying images, we need to make the inputs and outputs into images. We attempted to make a CAN learn how to construct plaintexts from ciphertexts. We tried 2 different methods for doing this:

1. Using randomly generated plaintexts, through both Salsa20 and the quarter-round function.
2. Using existing images, through only Salsa20.

Once we had encrypted and unencrypted images, we can train the network on them, and measure how well it learns. We varied multiple settings for the training to see how it affected the learning process:

- Patch size: From 16×16, 32×32. 64×64 and 256×256.
- Initial learning rate: From 0.000001 to 0.1.
- RGB weights: 1 to 3 for all RGB values, 1 to 32 for filters.
- Leaky ReLU (Rectified Linear Unit) Layers: 0.1 to 0.6.

Random plaintext-ciphertext pairs

This analysis method uses sets of plaintext (PT) ciphertext (CT) pairs, converts them into images and trains the CAN on them. See Figure 3.5 for an illustration.

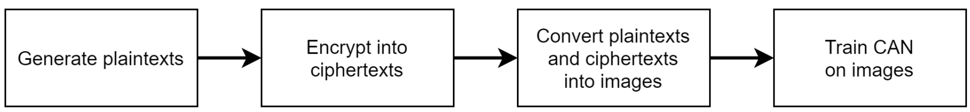

Figure 3.5: Algorithm for CAN analysis on PT-CT pairs.

In this approach, we generated 5 sets of binary pairs, each containing a million pairs. These are shown in Table 3.5.

When these sets of pairs were generated, they were converted graphical images. One set of images for the plaintexts and one set for the ciphertexts. To

Encryption algorithm	Size (bits)	Generation method
Salsa20	1024	Converted incrementally larger numbers $(0, 1, \ldots, 1000000)$ to binary. Padded to fit size requirement, and encrypted. See Figure 3.6 for a pair from this set.
Salsa20	1024	Generated random binary numbers of size 1024, and encrypted them. See Figure 3.7 for a pair from this set.
QR function	128	Generated random binary numbers of size 256, and encrypted them.
None	1024	Generated two random binary numbers of size 1024. No encryption. These are used for comparison to the encrypted pairs.
None	128	Generated two random binary numbers of size 256. No encryption. These are used for comparison to the encrypted pairs.

Table 3.5: Plaintext-ciphertext pairs generated for CAN analysis.

work well with the existing CAN setup, the images were made to be 256×256 pixel RGB images. This gave us 65536 pixel, each with 3 colors (red, green and blue), giving us a total of $256 \times 256 \times 3 = 196608$ values per image. We tried two methods of conversion:

- One bit per pixel. This gave us 65536 bits of plaintext/ciphertext per image, making it black and white. See Figure 3.8 for an example of one of these images.

- One bit per color per pixel. I.e. 3 bits per pixel. This gave us 196608 bits of plaintext/ciphertext per image file, making it a color image (RGB). See Figure 3.9 for an example of one of these images.

This gave us multiple plaintexts or ciphertexts per image, and the sets of images are therefore smaller than the sets of plaintexts and ciphertexts. Figure 3.10 shows how the system was set up, and Figure 3.5 shows an overview of the process.

Once we had a set of plaintext and ciphertext images, we could train the network on them.

As the Salsa20 algorithm consists of 80 quarter-rounds, we can expect the full algorithm to be more resistant to such analyses than a single quarter-round function. And since the random pairs are not related, we use these as

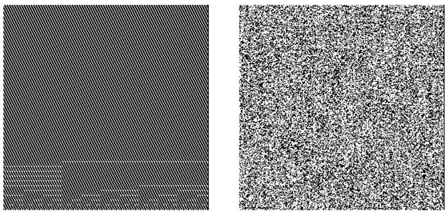

Figure 3.6: Example of an incremented PT-CT pair.

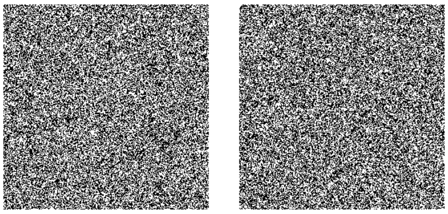

Figure 3.7: Example of a random PT-CT pair.

a null hypothesis and a base result for our tests. If the CAN learns no better on Salsa20 pairs or quarter-round pairs than random pairs, we can conclude that this method does not seem viable. If, however, there is a difference, we can conclude that these functions are at least somewhat weak against a CAN-based analysis. If, for some reason, the random sets have a significantly better learning rate than the other two sets, it may indicate that the PRF we use for generating the random pairs are weak against such analyses.

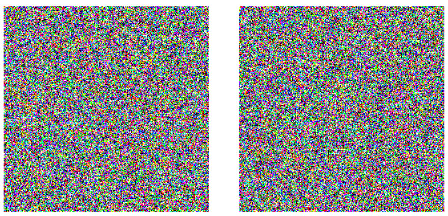

Figure 3.8: Example of a randomly generated PT-CT pair converted into RGB images.

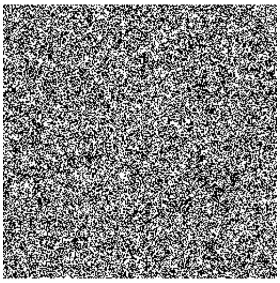

Figure 3.9: Example of a randomly generated plaintext converted into an black and white image.

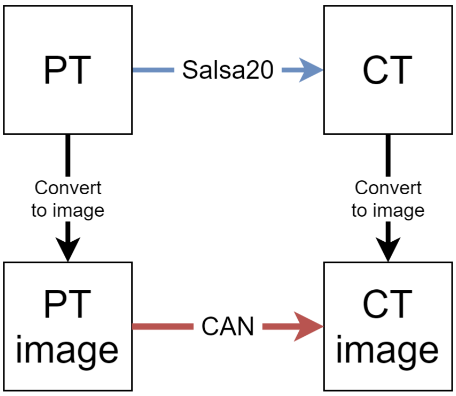

Figure 3.10: Illustration of our plaintext-based CAN analysis method.

Encrypted images

In addition to creating pairs of texts, we took existing images, reshaped them in Matlab into the required sizes for our CAN. We also encrypted them using the Python Salsa20 implementation. See Figure 3.11 for an overview of this method.

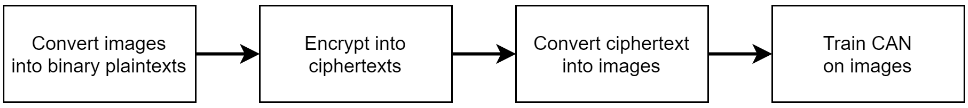

Figure 3.11: Algorithm for CAN analysis on PT-CT pairs.

That way, we use the original images as PT and the encrypted ones as CT. For an example of a image generate from the CT, see Figure 3.12 The

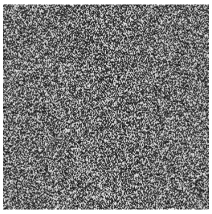

Figure 3.12: Image version of the CT from an encrypted image of a face.

analysis is otherwise the same as last section: Train the network on the CTs, so it will hopefully be able to construct their respective PTs. This method is illustrated in Figure 3.13.

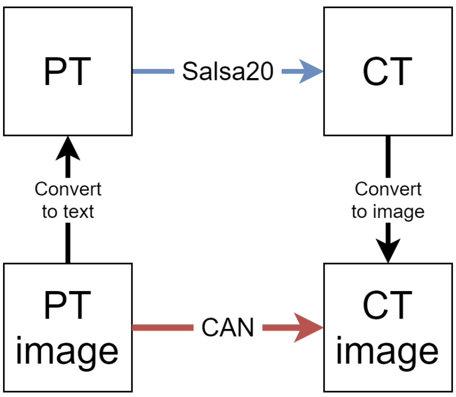

Figure 3.13: Illustration of our image-based CAN analysis.

Chapter 4

Results and discussion

4.1 Hamming distance

This section discusses the results from our differential analyses using the Hamming distance. First, we look at the results from Salsa20's quarter-round function, then we look at Salsa20's PRG function.

4.1.1 Quarter-round function

Salsa20 consists of 80 runs of the quarter-round function (QR), each of which progressively obfuscates the key more. If we define the QR function as:

$$QR(X) \rightarrow Y$$

then our analysis as modifying X into X' by changing some bits, and then measuring how the Y reacts. If $QR(X') \rightarrow Y'$, and the $HD(X, X') = n$, what is $HD(Y, Y') = m$? As we progressively incremented n, we saw m tend towards the expected equilibrium of half the length of Y. This analysis can be simplified to measuring:

$$HD\left(QR(X'), Y\right)$$

In Figure 4.1, the used values of X are 1024 bits. This is done to show the trends more clearly. While the QR function is made for 128 bits, it can also be applied to other amounts of bits. The x axis is in this case the amount of times the original X has flipped a bit, i.e., n. The y axis is the HD between the two values being measured, i.e., m. The lines in the legend are as follows:

0. $HD(X, Y)$
1. Convergence point of the Hamming distance between two random values.

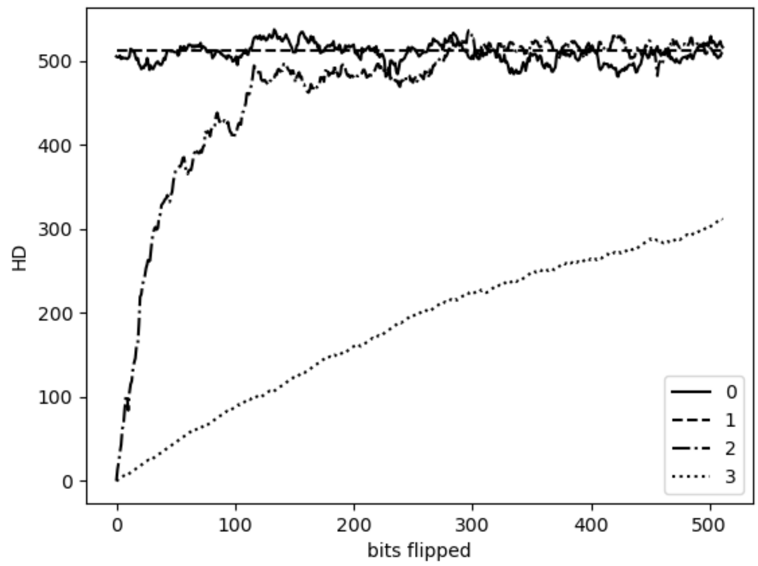

Figure 4.1: Effects of flipping random bits in X.

2. $HD(Y, Y')$

3. $HD(X, X')$

Line 0 and 1 show that the HD between an X and its corresponding Y is indistinguishable from the HD between two random values. In other words, the Hamming distance is not showing any correlation between inputs and outputs of the QR function. Line 3 shows, unsurprisingly, that the Hamming distance between two words increase at about 1 bit per random bit flipped. Line 2 shows that the Hamming distance between two Y values, where their corresponding X values are n bits distant from each other, drift towards the random/main sequence, or the expected value. It is however clearly distinguishable from random values, as long as X and X' are relatively close. In other words, as the $HD(X, X')$ is sufficiently small, the $HD(Y, Y')$ is smaller than what one can expect from random. This indicates a lack of sufficient avalanche effect in the quarter-round function. Assuming

$$HD(X, X') < \frac{\text{key size}}{8},$$

$HD(X, X')$ should be easily distinguishable.

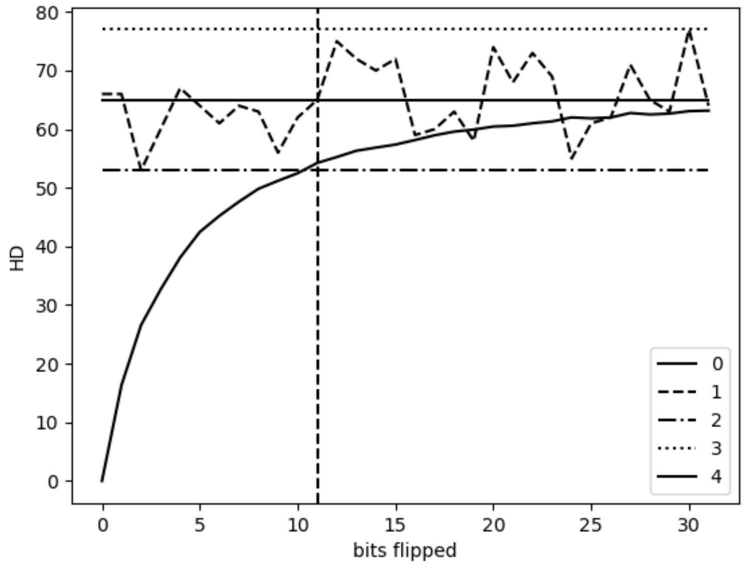

Figure 4.2: Averaging of effects on Y when bits are flipped in X.

In Figure 4.2, we see 5 non-vertical lines. The vertical line is where the bits flipped needed for the Hamming distance to be within the range of expected random distances. In this case, about 11 bits. In the figure, the lines represent the following:

0. $HD(X', Y')$ as n bits in X' are flipped.
1. $HD(X, Y)$ of random Xs.
2. Minimum expected difference between two random Xs.
3. Maximum expected difference between two random Xs.
4. Expected average difference between two random Xs.

4.1.2 Salsa20's PRG

We tried a similar analysis on Salsa20's PRG. In this case, we flipped random bits in the key, and saw how the output of the PRG reacted on the input. We can see the results in Figure 4.3. Line 1 shows how, unsurprisingly, how $HD(\text{in}_{\text{original}}, \text{in}_{\text{next}})$ is roughly equal to 1 per flipped bits. Line 2 is the expected value for random inputs and outputs. This is also the expected value

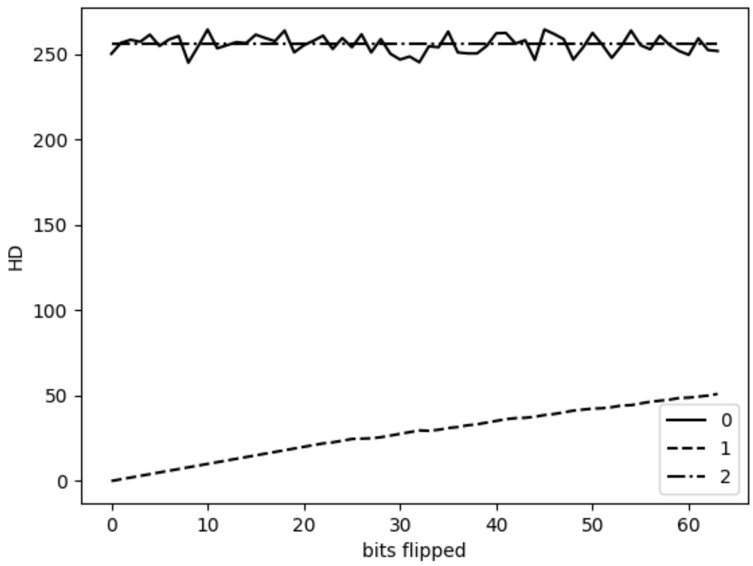

Figure 4.3: Averaging of effects on the PRG output when bits are flipped in key.

for an encryption algorithm with sufficient confusion. This is where we would expect the HD between random values to land. This is also where we would expect the HD between a good encryption algorithm's input and output to land. And when we look at line 0, this is indeed what we see. There seems to be no correlation or pattern between the amounts of bits flipped in the input (n) and the HD between the two values.

4.2 Context aggregation network

We made the CAN train on the 4 plaintext-ciphertext pairs, as well as the sets of graphical images. In Figure 4.4 we can see how the network trains on normal images, trending to 0 after a relatively short time.

For the random plaintext-ciphertext pairs, the network quickly stabilised the RSME and loss. However, it did not get close to 0, and remained on a much to high error rate. We can see this in Figure 4.5 for the images encrypted by the quarter-round function, and in Figure 4.6 for encryption by Salsa20.

The results for the image based analysis gave show as that, while the images gave us different results, the network was not able to train towards a sufficiently low value for use in cryptanalysis and attacks. In Figure 4.7, we

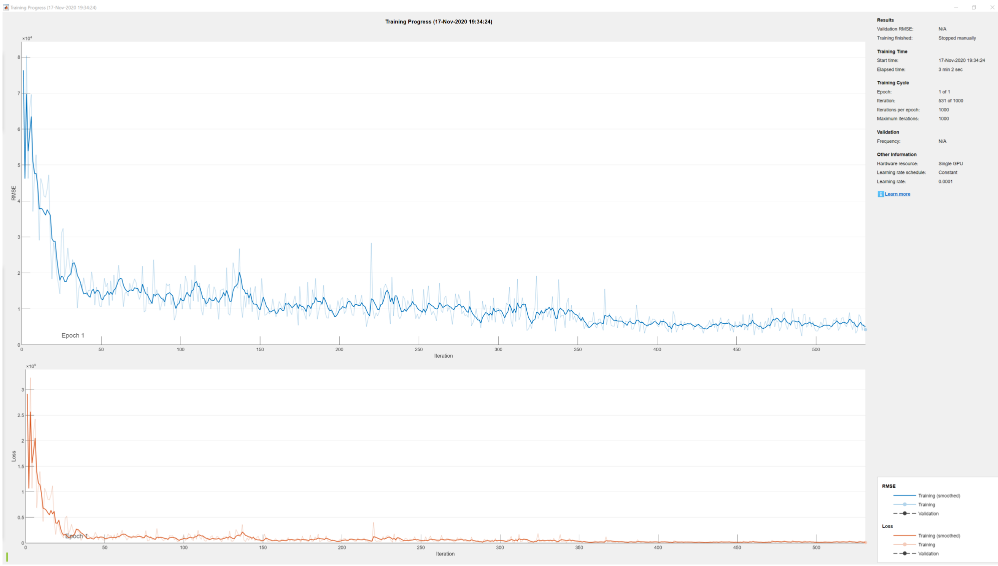

Figure 4.4: Training trend for regular images.

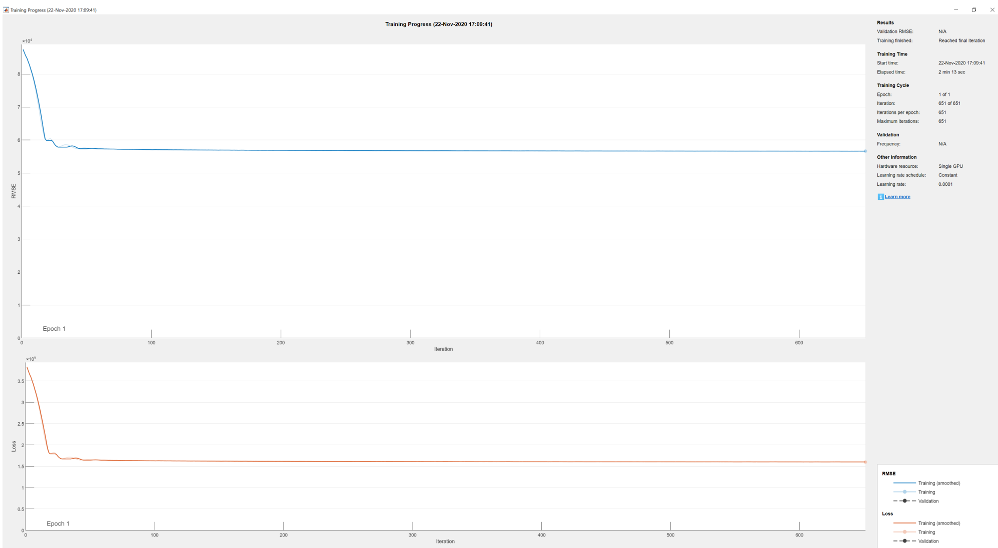

Figure 4.5: Training trend for the QR text pairs.

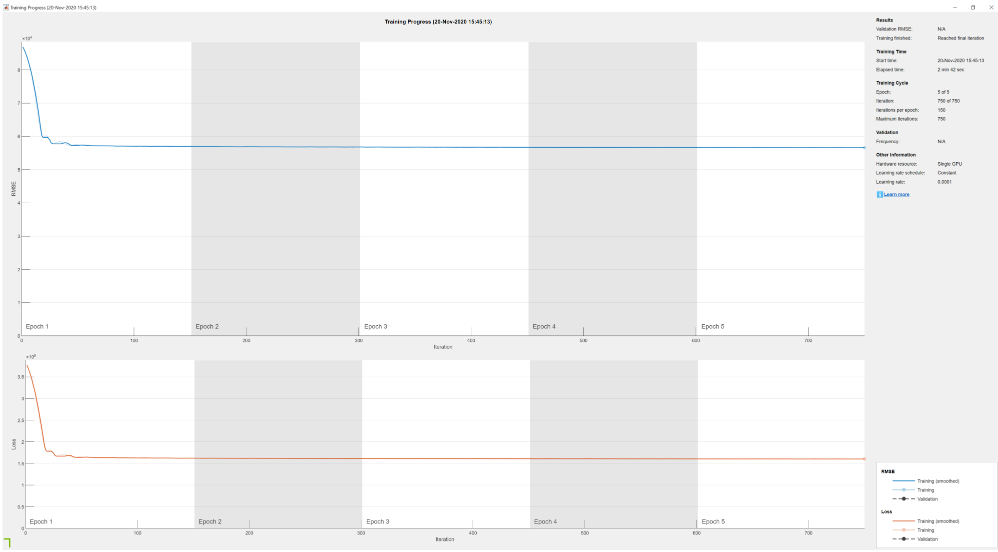

Figure 4.6: Training trend for the Salsa20 text pairs.

can see that the network also plateaued far from 0, when using regular patch sizes of 256×256. When using smaller patch sizes, we saw it still plateaued, but in a much more stable way. In Figure 4.8, we can see how the network trained on a 32×32 patch size. The result was similar when training overnight, about 300 epochs, 300000 iterations.

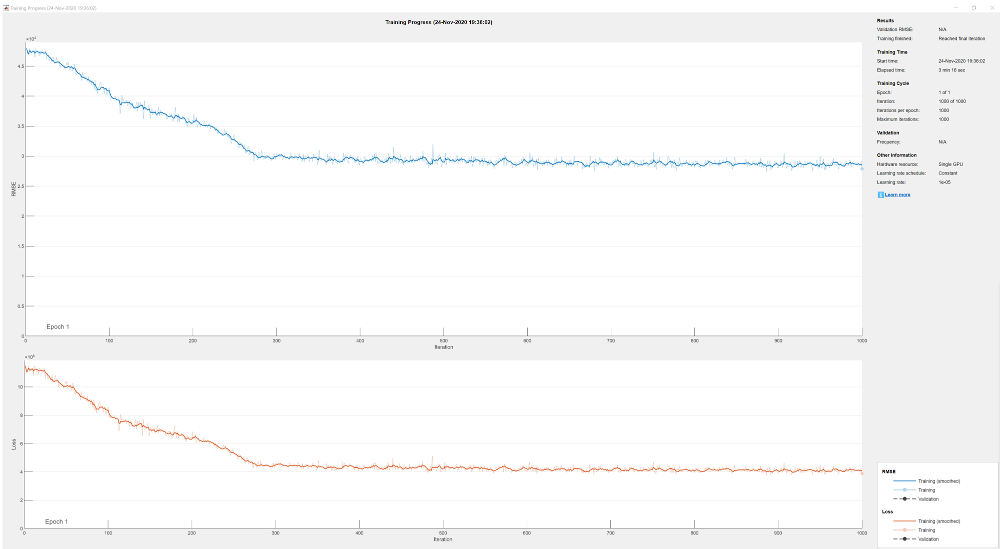

Figure 4.7: Training trend for the Salsa20 image pairs, with patch size of 256 × 256.

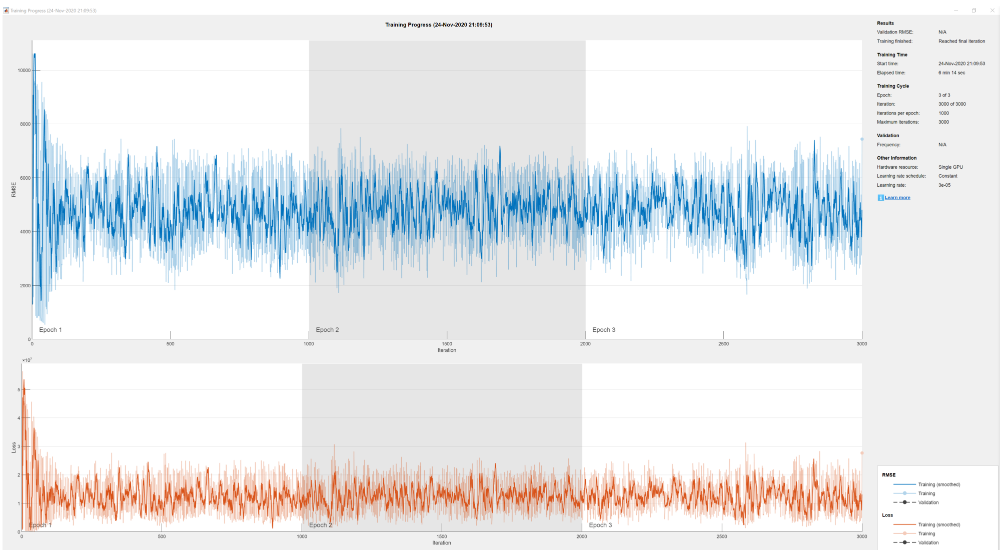

Figure 4.8: Training trend for the Salsa20 image pairs, with patch size of 32 × 32.

Chapter 5

Conclusion

This work attempted to answer the questions:

- Does Salsa20 seem to be secure against existing attacks?
- Can we break a weakened version of Salsa20?

Based on the found cryptanalysis research on Salsa20, there is little reason to think it is not sufficiently secure for normal use. There exists attacks more effective than brute-force, including side-channel attacks. Even so, these attacks are not practical, and so we conclude that Salsa20's security still holds up.

The analysis using the Hamming distance seem to show the quarter-round function does not have a sufficient avalanche effect to be an encryption algorithm by itself, but we also see that Salsa20 does not seem to suffer from the same effects. We therefore conclude that an attack based on this form of analysis, is not viable.

The analysis using the context aggregation networks shows little sign of effect. We therefore conclude that an attack using the CAN analysis we used, does not seem to have any feasibility.

5.1 Further work

While our context aggregation network did not show any signs of learning from the used data, it is possible that a modified loss function could work better. For example a loss function based on the Hamming distance.

As ChaCha and its quarter-round function is only slightly different, similar tests could be run at them to see if there is a significant difference.

Bibliography

[1] Daniel J. Bernstein. Salsa20 specification. 2005.

[2] Axel Poschmann. Lightweight cryptography - cryptographic engineering for a pervasive world. *ResearchGate*, 2009.

[3] Alex Biryukov and Léo Perrin. State of the art in lightweight symmetric cryptography. *University of Luxembourg/ePrint*, 2017.

[4] Muhammad Shoaib Siddiqui, Syed Obaid Amin, Jin Ho Kim, and Choong Seon Hong. A survey of wireless mesh networking security technology and threats. *IEEE Xplore*, 2007.

[5] Paul Gardner-Stephen, Romana Challans, Jeremy Lakeman, Andrew Bettison, Dione Gardner-Stephen, and Matthew Lloyd. The serval mesh: A platform for resilient communications in disaster crisis. *IEEE Xplore*, 2013.

[6] Anthony Gerkis. A survey of wireless mesh networking security technology and threats. *SANS*, 2006.

[7] Alex Biryukov. Block Ciphers and Stream Ciphers: The State of the Art. *The International Association of Cryptologic Research (IACR)*, page 22, 2004.

[8] Suhaila Omer Sharif and S.P. Mansoor. Performance analysis of stream and block cipher algorithms. *IEEE Xplore*, 2010.

[9] Yongqiang Li and Mingsheng Wang. Constructing s-boxes for lightweight cryptography with feistel structure. *Springer*, 2014.

[10] Nicky Mouha, Vesselin Velichkov, Christophe De Cannière, and Bart Preneel. Toolkit for the Differential Cryptanalysis of ARX-based Cryptographic Constructions. In François-Xavier Standaert, editor, *Workshop on Tools for Cryptanalysis 2010*, pages 125–126, Egham,UK, 2010. ECRYPT II.

[11] Vasileios Mavroeidis, Kamer Vishi, Mateusz D., and Audun Jøsang. The impact of quantum computing on present cryptography. *International Journal of Advanced Computer Science and Applications*, 9(3), 2018.

[12] Ray A. Perlner and David A. Cooper. Quantum resistant public key cryptography: A survey. In *Proceedings of the 8th Symposium on Identity and Trust on the Internet*, IDtrust '09, page 85–93, New York, NY, USA, 2009. Association for Computing Machinery.

[13] Saurabh Singh, Pradip Kumra Sharma, Seo Yeon Moon, and Jong Hyuk Park. Advanced lightweight encryption algorithms for IoT devices: survey, challenges and solutions. *Springer*, 2017.

[14] Tapalina Bhattasali. Licrypt: Lightweight cryptography technique for securing smart objects in internet of things environment. *University of Calcutta*, 2013.

[15] William J. Buchanan, Shancang Li, and Ramees Asif. Lightweight cryptography methods. *Journal of Cyber Security Technologies*, 2017.

[16] Sufyan Salim Mahmood AlDabbagh and Imad Al Shaikhli. Lightweight block ciphers: a comparative study. *NIST - National Institute of Standards and Technology*, 2015.

[17] Sergey Panasenko and Sergey Smagin. Lightweight cryptography: Underlying principles and approaches. *International Journal of Computer Theory and Engineering*, 2011.

[18] Thomas Eisenbarth, Christof Paar, Axel Poschmann, Sandeep Kumar, and Lefi Uhsadel. A survey of lightweight cryptography implementations. *IEEE Deisgn Test of Computers*, 2007.

[19] ECRYPT. eSTREAM: the ECRYPT stream cipher project. Accessed: 2020.05.26. URL: https://www.ecrypt.eu.org/stream/.

[20] CryptoLUX. Lightweight cryptography. Accessed: 2020.03.09. URL: https://www.cryptolux.org/index.php/Lightweight_Cryptography.

[21] Alex Biryukov, Dmitry Khovratovich, and Ivica Nikolić. Distinguisher and related-key attack on the full aes-256. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009*, pages 231–249, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[22] Sufyan Salim Mahmood AlDabbagh and Imad Al Shaikhli. Lightweight block ciphers: a comparative study. *Journal of Network and Computer Applications*, 2015.

[23] Alex Biryukov and Eyal Kushilevitz. Improved cryptanalysis of RC5. *Springer*, 2006.

[24] Nicky Mouha, Bart Mennink, Anthony Van Herrewege, Dai Watanabe, Bart Preneel, and Ingrid Verbauwhede. Chaskey: An efficient MAC algorithm for 32-bit microcontrollers. *Springer*, 2014.

[25] George Hatzivasilis, Konstantinos Fysarakis, Ioannis Papaefstathiou, and Harry Manifavas. Twine: A lightweight, versatile block cipher. *ResearchGate*, 2011.

[26] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clack, Bryan Weeks, and Louis Wingers. The SIMON and SPECK families of lightweight block ciphers. *NSA*, 2013.

[27] Bernhard Jungk and Shivam Bahsin. Don't fall into a trap: Physical side-channel analysis of ChaCha20-Poly1305. *IEEE Xplore*, 2017.

[28] Alexandre Adomnicai, Jacques J.A. Fournier, and Laurent Masson. Bricklayer attack: A side-channel analysis on the chacha quarter round. *ePrint*, 2017.

[29] Bodhisatwa Mazumdar, Sk Subidh Ali, and Ozgur Sinanoglu. Power analysis attacks on ARX: An application to Salsa20. *IEEE Xplore*, 2015.

[30] Alex Biryukov, Deike Priemuth-Schmid, and Bin Zhang. Multiset collision attacks on reduced-round snow 3G and SNOW 3G+. *University of Luxembourg*, 2010.

[31] Anaïs Querol Cruz. Conditional differential cryptanalysis of the post-quantum ARX symmetric primitive Salsa20. *Univeristé Denis Diderot Paris*, 2018.

[32] Nicolas Belleville, Damien Couroussé, Karine Heydemann, and Henri-Pierre Charles. Automated software protection for the masses against side-channel attacks. *Association for Computing Machinery*, 2018.

[33] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. *IACR*, 2013.

[34] D. A. Ferrucci. Introduction to "This is Watson". *IBM Journal of Research and Development*, pages 1–15, 2012.

[35] Wensheng Cheng, Wen Yang, Min Wang, Gang Wang, and Jinyong Chen. Context aggregation network for semantic labeling in aerial images. *Multidisciplinary Digital Publishing Institute (MDPI)*, 2019.

[36] Wensheng Cheng, Wen Yang, Min Wang, Gang Wang, and Jinyong Chen. Context aggregation network for semantic labeling in aerial images. *Remote Sensing*, 11(10):1158, 2019.

[37] Eric Rescorla. The transport layer security (TLS) protocol version 1.3.

[38] Abiodun Esther Omolara, Aman Jantan, Oludare Isaac Abiodun, and Humaira Arshad. An Enhanced Practical Difficulty of One-Time Pad Algorithm Resolving the Key Management and Distribution Problem. 2018.

[39] Christophe De Canniere and Bart Preneel. Trivium specifications. *eSTREAM, ECRYPT Stream Cipher Project*, 2006.

[40] Elena Andreeva, Begül Bilgin, Andrey Bogdanov, Atul Luykx, Bart Mennink, Nicky Mouha, and Kan Yasuda. Ape: Authenticated permutation-based encryption for lightweight cryptography. *Springer*, 2015.

[41] Daniel J. Bernstein. ChaCha, a variant of Salsa20. 2008.

[42] Loup's. The design of ChaCha20, 2017. Accessed: 2020.05.26. URL: http://loup-vaillant.fr/tutorials/chacha20-design.

[43] Paul Knutson. Cryptanalytic attack on Salsa20, 2020. Accessed: 2020.11.26. URL: https://github.com/catsymptote/Salsa_cryptanalysis.

Attachments

1. Implementation of Salsa and ChaCha, with analysis tools.