CPA-Attack on ChaCha20

Bastian Buck and P. Wilhelm Volz

Albstadt-Sigmaringen University of Applied Sciences, Albstadt, Germany,
{buckbast,volzphil}@hs-albsig.de

Abstract. The ChaCha20 stream cipher is considered a direct replacement for the AES block cipher and is often used, for example, in TLSv1.3. In addition to high speed and mostly small code size, it is considered secure. However, this only applies to typical attacks without considering power analysis side channel attacks. With such an attack it is possible to to extract the used key. To prevent this, two countermeasures come into question, which increase the security, but weaken the performance.

Keywords: ChaCha20 · CPA · Correlation Power Analysis · Side-Channel-Attack

1 Introduction

ChaCha is a family of stream ciphers based on the Salsa20 ciphers introduced by Daniel J. Bernstein [AFM17]. It is standardized by the IETF in RFC 8439. In this, the length of the counter and nonce is modified from the version presented in [Ber08]. ChaCha is increasingly used in current cryptographic protocols including TLSv1.2 and TLSv1.3 and various products such as the Wireguard VPN protocol, which is considered secure. Due to low complexity and lack of attack opportunities, this trend has remained intact to date and is expected to continue. The attack shown here is based on the CPA method, which functionally extracts the key but requires great effort. The correlation power analysis (CPA) attack belongs to the side-channel attacks. It can be used to extract secret keys from cryptographic devices. In this article, we demonstrate the application of a CPA attack to the ChaCha20 encryption algorithm. Using CPA, an attacker can analyze the power consumption or electromagnetic radiation emitted by a cryptographic device during encryption to extract the secret key used by the device. We first provide an overview of the ChaCha20 algorithm and how it is used to encrypt data. Then, we will explain how our CPA attack works and what types of side-channel information can be used for such an attack. We will also discuss the potential impact of a successful CPA attack on the security of the ChaCha20 algorithm and the measures that can be taken to protect against such attacks. Finally, we will evaluate the partially implemented countermeasures for their usefulness according to current statistical tests.

2 The ChaCha20 stream cipher

ChaCha20 is a stream cipher based on the 20-round cipher Salsa20/20. The changes from Salsa20/20 to ChaCha20 aim to improve the diffusion per round, which presumably increases the resistance to cryptanalysis, while maintaining - and often even improving - the time per round. In addition to the ChaCha20 version, there are also ChaCha12 and ChaCha8 with 12 and 8 rounds, respectively. These are useful depending on their use in algorithms where performance plays an essential role." [Ber08] [JB17] The ChaChaX family is considered an ARX cipher, which results in the use of AND, rotation, and XOR gates exclusively during execution [Ber08]. Important here is the differentiation between

Rotate ($\lll$) and Shift ($\ll$). For example, the byte with state [00 11 00 11] is given. A shift by 3 places to the left would produce the following result [11 00 00 00]. The bits are thus "shifted out" of the byte. A rotate, on the other hand, places the shifted bits at the end of the byte. To continue the example, the result for rotate would be the following: [11 00 11 00].

The initial state of ChaCha20 consists of 16 32-bit words, which are composed of constants (C), the key (K_{0-7}), a block counter (Counter, B), and a random string (Nonce, N) as shown in Table 1. The core of the ChaCha algorithm is the so-called "quarter round". It works with four unsigned 32-bit integers, which we have labeled a, b, c, and d. In Figure 1, the flow of a ChaCha Quarter Round is shown. ChaCha20, as mentioned, consists of 20 rounds executed alternately on a column and a diagonal of the state. In each round, the quarter-round function is executed four times in succession for each of the 16 words, each time generating four new values from the state [NL15].

After completion of the 20 rounds, the variable plaintext is linked to the key stream using XOR. Although a 64-byte block is generated with each round generation, only the amount of bytes that are also necessary for the encryption are used. In a nutshell, this means that a 65 byte plaintext would require 20 rounds twice. 64 bytes of the first key stream are needed for the first part, 1 byte of the next key stream for the remaining byte.

Table 1: Initial ChaCha20 state

C	C	C	C
K_0	K_1	K_2	K_3
K_4	K_5	K_6	K_7
B	N	N	N

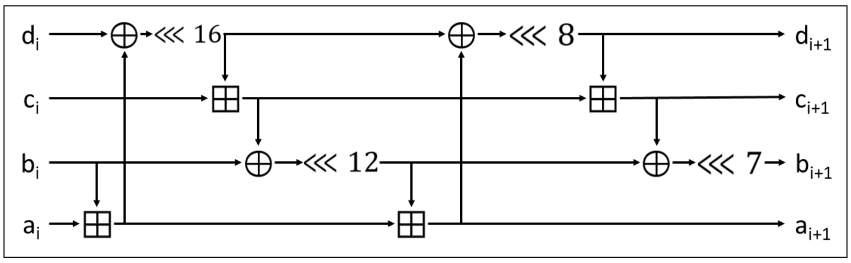

Figure 1: ChaCha Quarter Round

For the attack described in this paper, a ChaCha20 version written in the C programming language from shiffthq's public GitHub repository was used [isa16]. The code does not have any implemented countermeasures.

3 CPA attack model used for ChaCha20

Different approaches are known for a CPA attack on ARX ciphers. On the one hand, it is known that ARX ciphers are resistant to timing attacks, which is why such attacks are already ruled out. In [AFM17] and [JB17] two approaches are shown. Both possibilities use thereby the Divide & Conquer principle. In simplified terms, for 32-bit words $2^{32} \approx 4.294$ billion possibilities are possible for such a keyword. This seems too computationally intensive and impractical. Instead, the sources considered work byte-by-byte. This corresponds to $2^8 = 256$ possibilities for a key byte. For a 256-bit key, this would correspond to $256/8 = 32$ bytewords to figure out. [JB17] [Wu+08] [MAS15]

To classify the correlation between the values, the Pearson Correlation Coefficient is suitable. The required values are, on the one hand, the measurement points and, on the other hand, the power models of the hypothetical keybytes represented by the following equations. The return value of the Pearson Correlation Coefficient is always in the range +1 and -1, where the absolute highest value promises the best match. [JB17] [Jun22]

For the attack on ChaCha20 presented here, the Hamming weight (HW) model was chosen similar to that in [JB17]. The Hamming weight involves counting the bits represented in the bit representation. For example, the Hamming weight of [00 11 00 11] = 4, since 4 bits are set. Compared to the Hamming distance model, only the current state is important. If the previous byte changed to [01 00 01 00], the Hamming weight would be 2, but the Hamming distance would be 6, since 6 bits changed." [MOP08] [JB17] [Jun22]

To attack the key bytes $K_{0,\cdots,3}$ the following model (1) was used. Here + is equivalent to $x + y \mod 2^{32}$. [JB17]

$$HW(d_1) = HW(d_0 \oplus a_1) = HW(d_0 \oplus (a_0 + b_0)) \tag{1}$$

The value b_0 is the key byte to be attacked, which can assume the values 0-255 (byte) and is thus to be found by trying. The values a_0 and d_0 are values from the real used initial values. a_0 is a byte of a word of a constant and b_0 a byte of the block counter or the nonce (depending on the column/diagonal value). Consequently, these must be known. The only unknown is therefore the set of keybytes to find. Big- and little-endian must also be taken into account. The first byte attacked is the least-significant byte (LSB) of the first keyword. It should also be noted that the XOR used results in two mirrored equivalent values during the attack. These are then both in the negative as well as positive range with the same absolute value. [JB17]

The following model (2) was then theoretically used for the key bytes $K_{4,\cdots,7}$.

$$HW(b_1) = HW(b_0 \oplus c_1) = HW(b_0 \oplus (c_0 + \underline{\mathbf{d_2}})) = HW(b_0 \oplus (c_0 + (d_1 \lll 16))) \tag{2}$$

Practically, the rotation on the byte values proved to be complicating. Nevertheless, parts of the key could be used by means of the following alternative model (3). [JB17]

$$HW(b_1) = HW(b_0 \oplus c_1) = HW(b_0 \oplus (c_0 + \underline{\mathbf{d_1}})) \tag{3}$$

It should be noted that the rotation is omitted without alternative. Since the calculation is done on byte level, this has small effects, but they are still within the scope. This attack then uses the value b_0 from the first attack part, as well as for the value of c_0 the 256 possible key bytes in each case. The value d_1 is to be taken from the first attack part. The result now not only shows the correct keybyte for c_i, but also eliminates the wrong key hypothesis from the first attack part. Consequently, both key hypotheses are thus found by testing the correlation coefficients. [JB17]

3.1 Attack environment

The attack was carried out using ChipWhisperer hardware and software from the organization NewAE Technology Inc.[1]. For the programming part of the attack, on the one hand, a prepared virtual machine with Python 3.9, Numpy, Jupyter notebook, necessary compilers etc. is installed, on the other hand, the individual connections for the measurement by the used ChipWhisperer-Lite 32-bit board can be found here. This has among other things the necessary oscilloscope, in order to capture the measuring points.

Numpy was used as far as possible for the calculation of various values. Although the attack models can also be implemented in Python, there are large performance losses that can be eliminated by using optimized functions. The calculation of the Hamming weight is done by exclusive use of the operations Shift ($\ll$), AND ($+$), XOR ($\oplus$) and a multiplication ($\cdot$). The evaluation of the performance takes place in section 6.

The attack model and environment show the theoretical flow, the practical execution is related to concrete details and will be explained in the following chapter.

4 Execution of the CPA attack

For the practical implementation, the implementation of the ChaCha20 algorithm mentioned in section 2 was set up on the ChipWhisperer-Lite (32-bit) via a customized firmware. Connection-specific properties for the communication between computer and board were introduced. Measurements were then performed to varying degrees, each using a new value for the block counter and the nonce. Both values were transferred to the device in one and parsed by the firmware. Again, the endianness is of great importance. The number of measurements ranged from 500 - 50 000, where even 500 measurements gave good results and values of 50 000 could mitigate the natural noise. The measurements were cached in a format optimized for Numpy so that adjustments in the analysis did not require the entire measurement to be repeated. Measurements took from a few minutes for 500 measurements to ~ 1.37 hours for 50 000 measurements, depending on the number. The time depends on the connection to the board, in this case USB 3.0, as well as the hardware itself.

The measurement, as well as the following attack scenario, were mainly each generated and executed using a Jupyter notebook in the Python programming language. However, the use of Jupyter notebooks could lead to the case that default configured limits are reached or exceeded. In order to catch this case, the software code is additionally summarized in a simpler Python file.

The actual attack, as shown in section 3, is conducted using two different approaches, the latter depending on the result of the first part. As shown in Figure 1 and Equation 1, the initial values of the constants, the first keybyte and a part of the counter or nonce are used. In this case the following fixed values were used:

$$Konstanten : 0x61707865, ...$$

see in addition [Ber08].

$$Key : 0x00, 0x01, 0x02, 0x03, 0x04, .., 0x1d, 0x1e, 0x1f$$

$$Counter/Nonce : 0x00, 0x00, 0x00, 0x01 - - - 0x00, ...$$

The counter is in little-endian format equivalent to the decimal 1. The nonce is zeroed out here. In the attack scenario counter and nonce are chosen pseudo-randomly.

$$Klartext : 0x48, 0x65, 0x6c, 0x6c, ...$$

[1]https://wiki.newae.com/Main_Page

The fully used plaintext forms the sentence "Hello from Chipwhisperer, Im only here to get encrypted and you?", which corresponds to exactly 64 bytes, i.e. one encryption block.

The attack for the first keybyte (b_0) would then look like this:

$$HW(\text{0x64 (0b1100100)...}) = HW(0x01 \oplus (0x65 + b_0 \text{ (hier 0x00; für } b_0 = 0 - 255)))$$

For the second attack, the hypothesis b_0 (here 0x00) and the value of d_1 (here 0x64) would then be represented after Equation 3 as follows:

$$HW(\text{0x74 (0b1110100)...}) = HW(0x00 \oplus (c_0 \text{ (hier 0x10; für } c_0 = 0 - 255) + 0x64))$$

The examples show the attack for the first byte of the first keyword (of the first column round) in each case.

In order to carry out an automated attack, loops are necessary which, on the one hand, use 16 iterations, where each iteration corresponds to the attack on one byte of the cipher key. In each iteration, for each measurement (500 to 50 000) and each possible byte state (0-255), the Hamming weight of the intermediate encryption result is calculated. Then, the correlation between this and that of the measurement points is calculated. The best estimate for the current byte of the cipher key is determined as the byte value with the maximum/minimum correlation and stored in a list. Finally, the best estimates for the cipher stream (in hexadecimal format) are displayed.

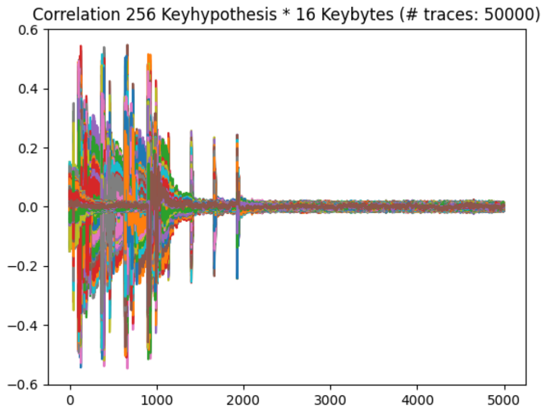

Figure 2: ChaCha CPA Attack Part1 (y = correlations; x = # trace-points per trace)

The calculation of the Pearson correlation coefficient can be clearly demonstrated with Figure 2 and Figure 3. Here, in Figure 2, the absolute correlation is always the same due to the XOR in the first attack part, resulting in the mentioned intermediate values. In Figure 3 this is no longer the case and shows a unique result for each key hypothesis from part 1.

In the first part of the attack, up to 10 (of 16) keybytes could be found successfully. In the second case, significantly fewer correct results are to be expected due to the required first correct keybyte. Here, up to 4 keybytes could be found. This may not seem very promising, but it is a mistake when looking at the bit level. Accordingly, a direct bit comparison shows a match of partly more than 80% (or up to 208 of 256 correct keybits). Mostly bit skips are decisive for the fact that the keybyte is not correct. Possibly, however, more extensive measurements could solve this problem.

Thus, it could be shown that a CPA attack on the ChaCha20 stream cipher is efficiently possible within minutes even with a few measurements. To make the used variant secure against such an attack, we will explain possibilities for countermeasures and present and evaluate a practical implementation in the following chapter.

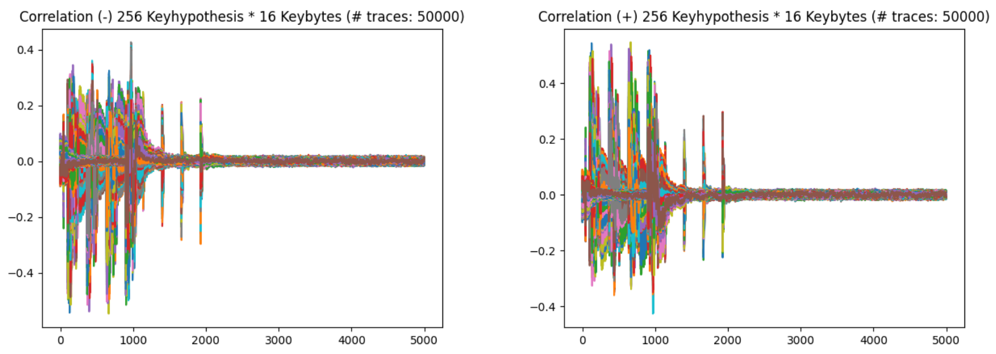

Figure 3: ChaCha CPA Attack Part2 (negative (<) & positive (>) b_0 key-hypothesis) (y = correlations; x = # trace-points per trace)

5 Hardening of the ChaCha20 implementation

In principle, two types of countermeasures are possible. Hiding disturbs the signal-to-noise ratio. The noise is increased and the correlation between the values is reduced. Masking, on the other hand, ideally masks the intermediate values completely so that an attack is no longer possible. However, a perfect variant is practically impossible." [JB17] Randomness is needed for each of the countermeasures. This is done in this project using pseudorandom numbers. There are options besides the $rand()$ function in C such as the Mersenne Twister or optimized PRNG such as the XorShift128+ used in this project (see [Vig16]). This uses two 64 bit numbers as seed and thus generates a pseudo-random 64 bit number. This can then be used for the respective purposes. In the present case, the seed corresponds to the first two words of the nonce. However, this must not be applied to a practical implementation.

5.1 Masking of the Quarterround

Conceptually, an effective countermeasure could be masking the internal states of the quarterround function using XOR. As mentioned, only operations such as ADD, Rotate and XOR are used in this function. Masking by ADD or MUL would be unnecessarily difficult and would have losses in the performance. For masking XOR a random bit would have to be generated for a bit to be masked. For rotate, no separate random is needed. However, for masking the addition modulo 2^{32} there is no trivial possibility is available. For this, the Trichina AND gate can be used, but this also means another random bit for each AND and requires four intermediate values. [Jun22] [MOP08] Due to this barrier, no implementation was done in this work.

5.2 Hiding: Shuffling of the execution order of the quarterrounds & Dummy Instructions

Hiding measures can take various forms. In the present case, shuffling and dummy instructions are easily implemented and can be of great use. The C implementation of ChaCha20 mentioned in section 2 now has to be hardened by means of a combination of these same hiding measures, so that the noise makes evaluation non-trivially possible with a small number of measurements.

Shuffling means the random sequence of function calls, so that the final value nevertheless corresponds to the correct counterpart. Dummy instructions are inserted between the respective function calls so that they cannot be trivially filtered out. At the same time, these have no functionality to the actual cryptoalgorithm. [Jun22]

We have achieved an entanglement between shuffling and dummy instructions, such that many dummy instructions are randomly executed in various orders on each run. The principle is built into the ChaCha20 block function and uses the mentioned XorShift128+ PRNG (see [Vig16]) for generating pseudorandom numbers. Since the counter/nonce is modified on each call, prior automatic filtering is not present. Our algorithm uses the following flow within the block function for both column and diagonal rounds:

1. Create array *done* with intermediate values and target value y
2. As long as target not reached:
    (a) Generate random value x between 0 and # rounds.
    (b) Generate new random numbers rnd_i in the range 0 and 3 until x is reached.
    (c) Check if execution of the round (Column(0-3) or Diagonal(0-3)) has already been recorded in *done*.
    (d) If already executed, start over from item 2; otherwise execute round.
3. Check if all rounds are executed; if yes, set target value y, otherwise continue.

It is trivial to see that, depending on the round functions already called, the probability of obtaining the correct random number for the next round function becomes increasingly low. Consequently, there are strongly increasing calls to dummy instructions, which can be adjusted depending on the performance requirements in the item 2a area for the number of rounds. It has been shown that even omitting these additional dummy instructions makes the utilization of low measurement points very difficult. More on this in section 6.

6 Evaluation of ChaCha20 hardening

After implementing the hiding measures, the same attack was performed as for the unprotected version. Again, several measurements of different sizes were created and analyzed. The plots of the attack on the unprotected implementation (Figure 2 and Figure 3) show the functionality of the countermeasure in comparison to the plot of the attack on the hardened implementation (Figure 4). It is very clear to see that no or only a few correlations (seen in the form of "peaks") can be found here. This is further confirmed in the output of the attack. Here, none of the key bytes used could be correctly determined. A measurement with 50,000 measurements showed the same result, but was able to identify the first more concrete peaks in the range of the first 1,000 measurement points of different measurements.

6.1 Attack on the hardened ChaCha20 implementation and comparison

The same software code was used to perform the attack on the hardened implementation as for the unprotected implementation. Thus, the flow of the attack is the same in both cases.

By inserting the dummy instructions and mixing the relevant quarterround function calls of the ChaCha20 algorithm, the signal noise is amplified to such an extent that the determination of the expected value can no longer be performed accurately. This also becomes clear when looking at the graphical representation of the measurements, as seen in Figure 5. In the left diagram (without countermeasures), a clear pattern can be seen, while in the right diagram it is much more difficult to identify a pattern. In order to continue the

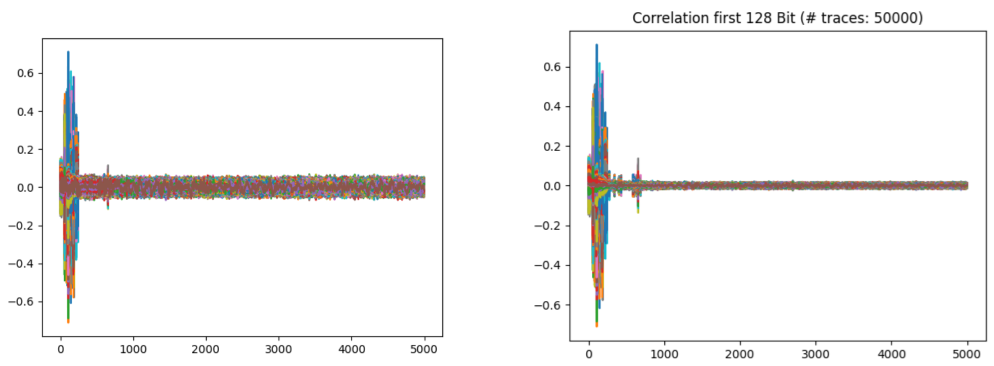

Figure 4: Shuffled CPA Attack 500 (<) and 50k (>) Traces Part1 (y = correlations; x = # trace-points per trace)

attack successfully, the number of measurements must be increased to sufficiently mitigate the noise. At a certain point, however, problems in the processing of the measured values, as well as the processing time, are to be expected. While these problems can be fixed, they require a new attack approach that incorporates them.

Another approach could be to filter out the dummy instructions and predict the randomization of the ChaCha20 instructions. The filtering of dummy instructions is comparatively easy to achieve. For this purpose, repeating always the same measurement ranges can be used as clues. For the prediction of the randomly executed instructions, the resolution is more difficult and can only be realistically implemented if a poor random generator was used for the randomization. In the present case, the PRNG (XorShift128+) used is sufficiently secure, but uses the first two 32-bit words of the nonce as the initial seed. Consequently, an attacker in possession of the nonce can predict the complete random stream and thus filter out executed dummy instructions. In addition, the sequence of calls can then also be predicted and the attack adapted accordingly.

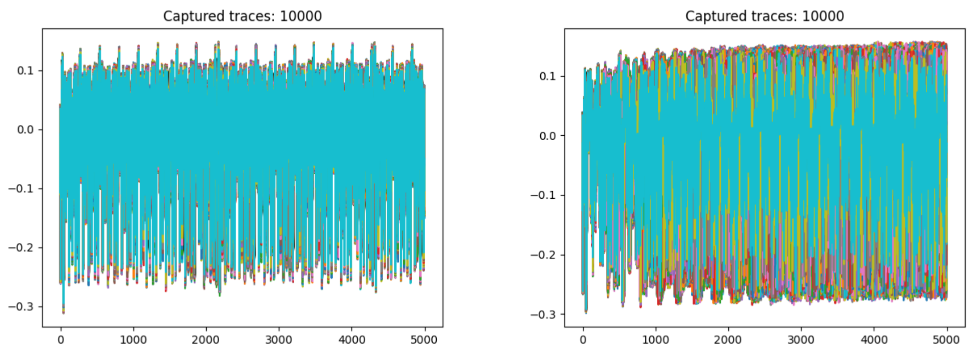

Figure 5: Measured 10 000 traces (no countermeasure (<) & hiding (>))

6.2 Statistical leakage test

To test the significance of the measurements, Welch's t-test can be used. For this purpose, measurements with permanently fixed values are compared with those of randomized inputs. Here, the null hypothesis between $[-4.5, 4.5]$ is accepted (no information tends to be obtained in this range). Values below and above this could thus provide information indicating the possibility of a working attack. [JB17] [Jun22] The statistical analysis performed using Welch's t-test shows that the normal implementation of the ChaCha20 algorithm exhibits very significant uniform leakage (see Figure 6 on the left). Leakage is also still evident for the hardened variant (see Figure 6 right). Here, however, extraordinarily high values show up at the beginning. The following values again show magnified similar values. This is the reason why the attack on these ChaCha20 implementations generally works or can work in the latter case.

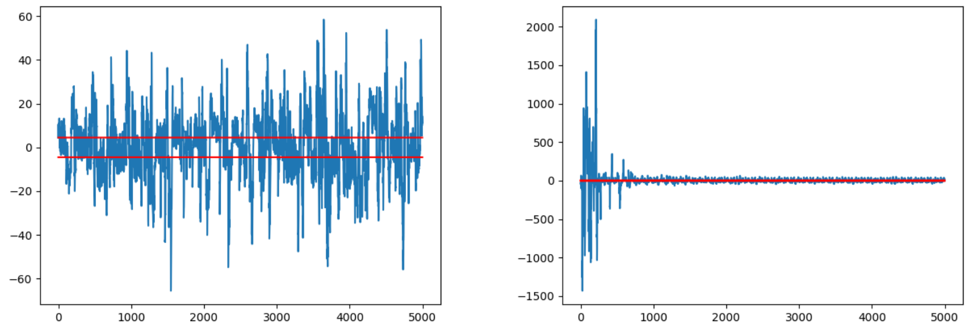

Figure 6: Welch't t-test with 500 traces (without countermeasure (<) & hiding (>)) (y = leakage; x = # trace-points per traces)

7 Conclusion

In conclusion, it can be said about the attack on the ChaCha20 algorithm that it is quite possible and even successful to a large extent. With an extraction of $\sim 83\%$ of the key bits, a significant step has already been made and it is thus clearly understandable that there is a vulnerability to a CPA attack. In order to find the remaining key bits, further ideas could come into play, the implementation of which would have to be examined in further work. Another option could be to use a different attack model. Possibly the use of the Hamming distance model is advantageous in this case. Also, an analysis of the generated assembly code may lead to further insights that can be used to optimize the attack.

In order to work with a larger number of measurements, it is necessary to design the calculation of the Pearson correlation coefficient for this use case. For this purpose, the algorithm proposed by Socha et al. in [Soc+17] for the iterative calculation of the Pearson correlation coefficient can be used.

References

[AFM17] Alexandre Adomnicai, Jacques JA Fournier, and Laurent Masson. *Bricklayer attack: a side-channel analysis on the ChaCha quarter round*. Springer, 2017. URL: https://link.springer.com/chapter/10.1007/978-3-319-71667-1_4 (visited on 02/07/2023).

[Ber08] Daniel J. Bernstein. *ChaCha, a variant of Salsa20*. 2008. URL: http://cr.yp.to/chacha/chacha-20080128.pdf (visited on 02/04/2023).

[isa16] isayme. *ChaCha20 stream cipher in C*. 2016. URL: https://github.com/shiffthq/chacha20 (visited on 02/06/2023).

[JB17] Bernhard Jungk and Shivam Bhasin. *Don't fall into a trap: Physical side-channel analysis of chacha20-poly1305*. IEEE, 2017. URL: https://ieeexplore.ieee.org/abstract/document/7927155 (visited on 02/07/2023).

[Jun22] Bernhard Jungk. *Vorlesung: Implementation Attacks and Countermeasures (WiSe 22/23)*. 2022.

[MAS15] Bodhisatwa Mazumdar, Sk Subidh Ali, and Ozgur Sinanoglu. "Power analysis attacks on ARX: an application to Salsa20". In: *2015 IEEE 21st International On-Line Testing Symposium (IOLTS)*. IEEE. 2015, pp. 40–43.

[MOP08] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*. Vol. 31. Springer Science & Business Media, 2008.

[NL15] Y. Nir and A. Langley. *ChaCha20 and Poly1305 for IETF Protocols*. 2015. URL: https://www.rfc-editor.org/info/rfc7539 (visited on 02/05/2023).

[Soc+17] Petr Socha et al. "Optimization of Pearson correlation coefficient calculation for DPA and comparison of different approaches". In: *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. IEEE. 2017, pp. 184–189.

[Vig16] Sebastiano Vigna. *PRNG: XorShift128+*. 2016. URL: https://xoshiro.di.unimi.it/xorshift128plus.c (visited on 02/08/2023).

[Wu+08] Keke Wu et al. "Correlation power analysis attack against synchronous stream ciphers". In: *2008 The 9th International Conference for Young Computer Scientists*. IEEE. 2008, pp. 2067–2072.