Universiteit
Leiden
The Netherlands

Bachelor Computer Science

Power leakage of

AES implementations

Niels de Wit

First supervisor and second supervisor:
Nusa Zidaric & Abolfazl Sajadi

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

20/08/2024

Abstract

This thesis investigates the power leakage of various AES implementations. The method for measuring leakage is Test Vector Leakage Assessment. This method is able to show the specific points in time where leakage occurs during encryption. Comparing the various implementations during encryption show that masking, bitslicing, and the counter mode of operation all reduce the leakage.

Table of Contents

Contents

1 Introduction

Recently, the European Council (EUCO) decided on updating their current cybersecurity strategy, due to the increasing level of cybersecurity threats. The EUCO strives to alert its Member States to safeguard a secure cyberspace for the population. [6] With security and privacy of information becoming more valued, there is an increasing need for encryption. To secure data there are processes of transforming this data which hides the original content. Ideally, only authorized parties would be able to decrypt the data back into its original form. One of these methods of transforming data is by using the Advanced Encryption Standard (AES). AES is a variant of the Rijndael block cipher[8]. Block ciphers are algorithms that handle data of a fixed length, called blocks.
This thesis aims to compare various AES implementations in order to check the presence of side-channel leakage. Exploiting this leakage, which is indirect information leaked during the execution of a program, allows the discovery of items such as the key used for AES encryption. The AES implementations examined are Tiny-AES[1], Masked-AES[2], PQClean[3], and T-Tables[4]. The AES modes of these implementations include Electronic Code Book, Cipher Block Chaining, and Counter. First, static and active analysis of the programs are applied to gain an initial view of the performance of the implementations. Then, power measurements are collected and analysed by using Test Vector Leakage Assessment (TVLA). These measurements are obtained using a ChipWhisperer device allowing easy analysis of various implementations. Finally, power analysis is performed using TVLA after which the outcome is evaluated. None of the implementations are capable of fully preventing information leakage from occurring, however, some implementations are able to decrease the amount of information leakage.

2 Background

First AES and some of the different modes it may use will be introduced in Section 2.1. Secondly, Side-channel analysis will be explained with the emphasis on power analysis and TVLA in Section 2.2. Lastly, the used device for obtaining measurements will be explained in Section 2.3.

2.1 AES and modes of operation

The Advanced Encryption Standard (AES), is a standardized method of encrypting plaintext via the usage of a secret key[10]. The Rijndael block cipher[8] was chosen as this standard by the National Institute of Standards and Technology (NIST)[10]. In this thesis the following AES modes will be used: Electronic Code Book, Cipher Block Chaining, and Counter mode.

2.1.1 AES

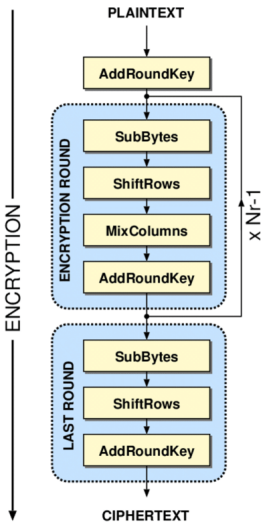
AES works by encrypting individual blocks of data using a substitution-permutation network (SPN). This network is the manner in which the data is encrypted, using substitution and permutation

[1] https://github.com/kokke/tiny-AES-c
[2] https://github.com/CENSUS/masked-aes-c/blob/main/aes.c
[3] https://github.com/PQClean/PQClean/blob/master/common/aes.c
[4] https://github.com/pjok1122/AES-Optimization/blob/master/aes.c

methods, as is seen from the steps in Figure 1. The blocks of data are 16 bytes large and can be organised in a 4×4 matrix, with 1-byte large cells. This matrix is also called the "State".
An overview of the encryption rounds can be seen in Figure 1, showing the main functions.

The cipher for AES is composed of a differing amount of rounds (N_r) depending on which keysize is chosen. For AES-128 N_r is set to 10. Before entering the cycle of rounds, the inserted plaintext and initial roundkey are XORd with eachother, also called key whitening. Next, the text goes through the encryption loop from Figure 1. Every function in this figure is further shown at the end of this section.

Figure 1: AES Rounds

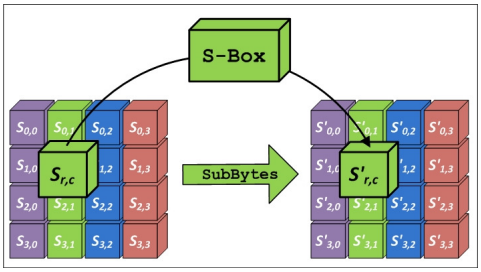


Source:Arrag[1]

These rounds work as follows: First, every byte in the 16-byte text block is substituted with its corresponding value from the S-box, Figure 2. Then, once all the bytes are substituted, the rows of the matrix will be rotated, Figure 3. The first row is left the same, the second is shifted once to the left, the third twice to the left, and the fourth is shifted thrice to the left. This can be simplified to rotating the rows the amount equal to the row index to the left. Next, the resulting matrix is multiplied with the matrix from Table 1 as in Figure 4. This is called the MixColumns step. Lastly, the roundkey is XORd with the resulting matrix, as in Figure 5.
The last round of every AES encryption omits the MixColumns step, yet still applies the S-box, row shifting, and addition of the roundkey.
The resulting ciphertext after all these steps is the fully encrypted plaintext using AES.
For the key expansion an initial key is required. This key is used to generate the multiple roundkeys (N_r) used in the encryption loop. The expanded key size for AES-128 is 176 bytes. The first 16 bytes are for key whitening. The other $N_r \cdot 16$ bytes are used at the end of every round.

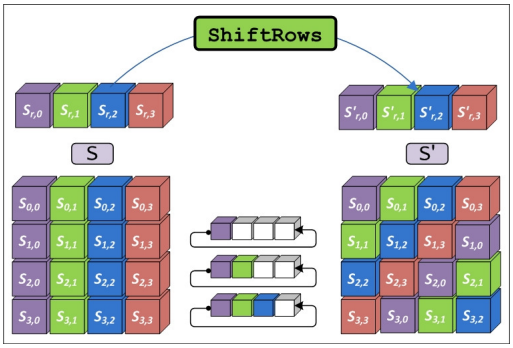
Expanding the key takes several steps. At first an initial 16-byte key is required and arranged in its 4×4 matrix form. Columns in positions that are a multiple of 4, the columns that start a new roundkey, are calculated by rotating the previous column one cell upwards. Next, each cell is substituted according to the used substitution box in the AES algorithm. For the last step, XOR the result with the column 4 positions earlier, the starting column of the previous roundkey, and XOR with a round constant. The other columns are calculated by applying XOR to the previous column, and the column 4 positions earlier. This is repeated until the required amount of roundkeys are made for completing all N_r rounds of encryption.

While AES is mathematically defined over the finite field GF(2^8), for the purposes of this thesis, knowledge of finite fields and mathematical specifications of AES are not required.



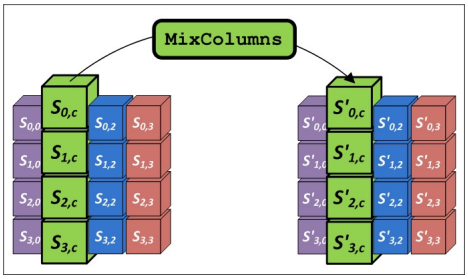
Source:Arrag[1]

Figure 2: SubBytes



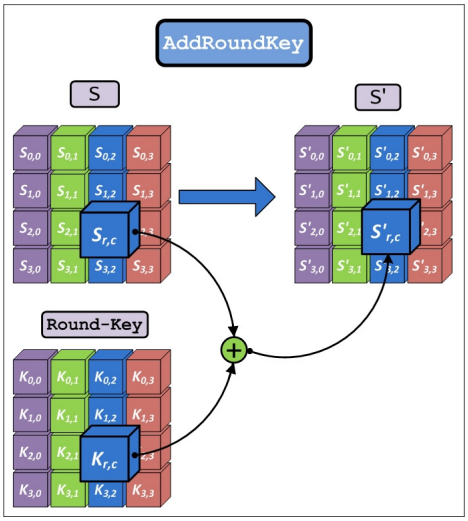
Source:Arrag[1]

Figure 3: ShiftRows



Source:Arrag[1]

Figure 4: MixColumns



Source:Arrag[1]

Figure 5: AddRoundKey

02	03	01	01
01	02	03	01
01	01	02	03
03	01	01	02

Table 1: MixColumns table

2.1.2 Modes

Next, the different used modes will be explained according to the NIST recommendations[9]. A brief overview will also be shown of various other modes. There will be a primary focus on differences within encryption. Decryption is not the focus of this thesis and will therefore not be elaborately

discussed, but will be included for completion.

Electronic Code Book (ECB) works by individually encrypting each block of 16 bytes, as visualized in Figure 6. The blocks are encrypted independently of each other. This may increase speed by parallelization, however, it is less effective in hiding patterns within larger plaintexts. If two blocks happen to be the same, and are encrypted with the same key, the resulting ciphertext will also be identical. Therefore, the total resulting ciphertext of the large plaintext is not fully illegible.

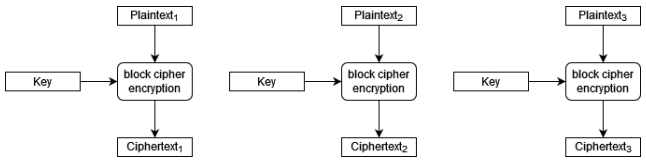


Figure 6: ECB mode encryption

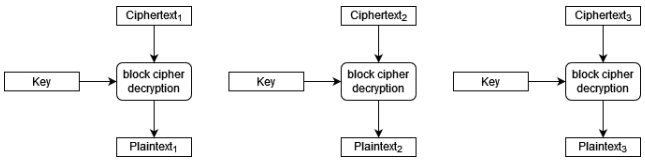


Figure 7: ECB mode decryption

Cipher Block Chaining (CBC), visualized in Figure 8, works by XORing the plaintext with an initial vector (IV). This resulting text is put through the block cipher. The calculated ciphertext is then XORd with the next block of plaintext in place of the IV. These steps are repeated until the entire plaintext is processed. The benefit of this method of encryption is that, contrary to ECB, there is no clear visible pattern when encrypting two identical blocks with the same key. This is because the IV XORd with the plaintext before encryption leads to a different result between the otherwise identical plaintext blocks. The disadvantage however is that this mode can not be parallelized. This is because the next block of plaintext can only be encrypted once it is XORd with the result of the previous block.

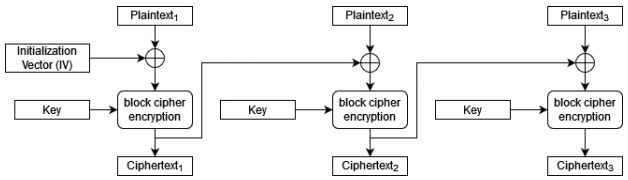


Figure 8: CBC mode encryption

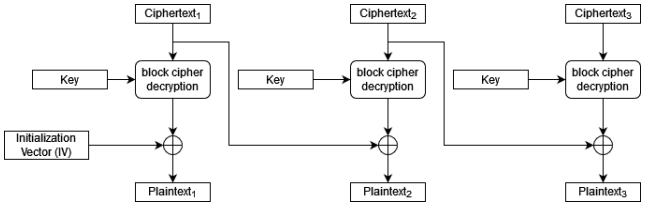


Figure 9: CBC mode decryption

Compared to the other modes used in this thesis, CBC is the only mode which supports Message Authentication Codes (MAC). The final block of output when performing CBC would in this case be the MAC. A MAC may be used to authenticate and verify the integrity of a message. This is because the previous (encrypted) plaintext is used for determining the output of the next ciphertext. If the ciphertext is tampered with in any way, be it accidental (transmission errors) or on purpose (malicious actor), this can be verified by entities who know the key by comparing the resulting MACs. If the MAC differs, the message was altered.

Counter mode (CTR) combines the benefits of CBC and ECB mode by allowing parallelization and identical plaintexts resulting into different ciphertexts because of a counter. This mode can

be seen in Figure 10. A nonce is inserted into the cipher instead of the plaintext. The resulting ciphertext is then XORd with the corresponding block of plaintext. As its name suggests, for the next block of data to be encrypted, the nonce is incremented before being encrypted. The result is again XORd with the corresponding plaintext and repeats until everything has been encrypted. The benefit of this method is that identical blocks of plaintexts do not result in identical blocks of ciphertext due to the incrementing nonce. This mode can also be parallelized as long as the position of the block of plaintext is known. The counter can simply be incremented by its position to then be inserted into the block cipher.
The XOR of the plaintext with a keystream is a one-time pad (OTP). OTPs can't be broken by an outsider if they adhere to the following requirements[16]: The key XORd with the plaintext is random, the key is never re-used and secret, and the key is as long as the plaintext. The CTR-mode implementation makes use of the properties of OTP if the nonce and key are handled properly.

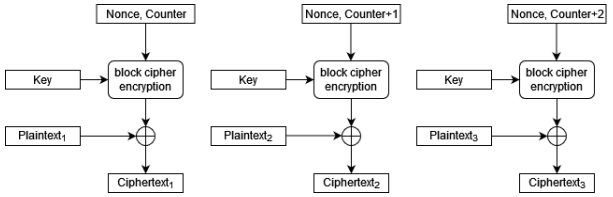


Figure 10: CTR mode encryption

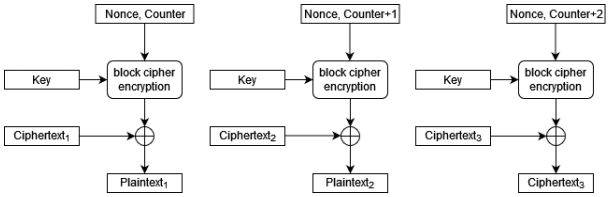


Figure 11: CTR mode decryption

There are various other modes that can be used for encryption. One of these modes is Output FeedBack (OFB), displayed in Figure 12. It XORs the plaintext with the keystream like the CTR mode again being a OTP, and chains the keystream with upcoming blocks like CBC mode. The chained keystream for OFB does not however involve the plaintext like CBC, only the IV and the key. The chaining works by inserting an IV into the cipher and routing that output as input for the next cipher leading to the feedback. After the output is rerouted it is XORd with the plaintext. This means that the plaintext does not influence the outcome of any other blocks.
Cipher FeedBack (CFB) is a mode very similar to OFB, displayed in Figure 14. The difference between these two modes is the value of the feedback: in the OFB mode the feedback is the output of the block cipher, ie the keystream, while in the CFB mode the feedback is the ciphertext, ie one-time pad of the plaintext and the keystream. This means that any tampering with the plaintext cascades into changes to the last block. Due to this cascading effect this last block can be used to verify integrity, functioning as a MAC, just like with CBC mode. These last two modes are also commonly implemented but will not be used in this thesis.

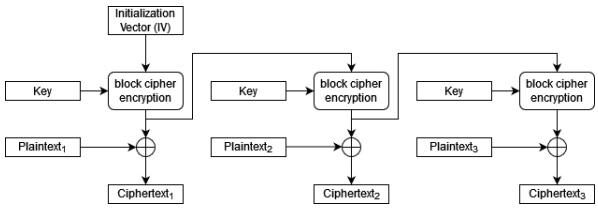


Figure 12: OFB mode encryption

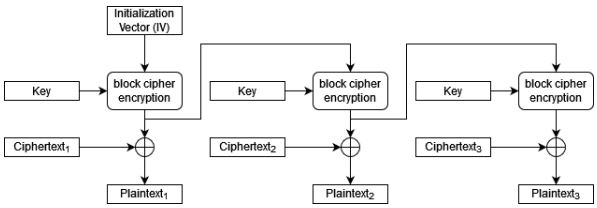


Figure 13: OFB mode decryption

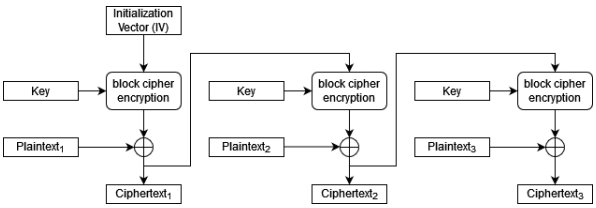


Figure 14: CFB mode encryption

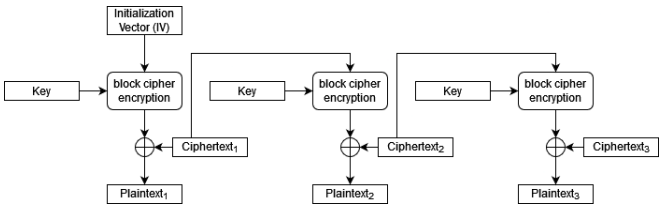


Figure 15: CFB mode decryption

2.2 Power Side Channel Analysis

Side-channel attacks exploit the indirect information that is leaked during the execution of a program. This indirect information is a side-effect when using a program in practice, as information can be gained by for example analysing power consumption. There are various ways to analyse this leakage.

2.2.1 General overview

There are various types of side-channel attacks. Timing attacks attempt to retrieve a secret parameter by analysing differences in execution time[26]. Electromagnetic attacks measure the emitted electromagnetic radation from the target device when executing the program being attacked[19]. Power analysis exploit the power consumption of the device. This thesis focuses on power analysis attacks.

Power analysis attacks are based on the assumption that the power consumption and operations being executed are correlated [14]. In order to perform this type of attack the power consumption of the device is measured. Given enough data, a pattern between the executed operations within a program and power consumption may be found, leading to the possibility of revealing the encryption key. Within this type of attack there are a few variants. Simple Power Analysis (SPA), Differential Power Analysis (DPA), and Correlation Power Analysis (CPA).
These types of attacks work by inspecting power traces. These traces contain measurements of the power consumption which are taken periodically during the execution of a cryptographic algorithm. SPA works by visually observing the power trace of a cryptographic algorithm and deducing which operations are occurring at a given point in time[13]. Closely observing one power trace reveals slight deviations in behavior, showing the execution path. Kocher et al.[13] suggests to avoid using secret intermediates for branching operations, such as the encryption key, which would show differences in the power graph.
DPA uses statistical methods instead of manual recognition to find links within the power graph of the cryptographic algorithm. The differences between the gathered samples are calculated by splitting the traces in subsets where the difference between the averages of these subsets is calculated. According to Kocher et al.[13] this method would allow any association within the algorithm to be found from the power measurements, given that enough traces are used in the calculation.
CPA tries to find a correlation between the predicted output and the power consumption of the device. One way to do this is by finding the correlation between power samples and the Hamming weight of the used data [4]. The Hamming weight of data is the amount of symbols that are not the zero-symbol. For the binary string `1101` the Hamming weight would be 3. Hamming weight can

be used to obtain hypothetical values. CPA is based around the idea that different keys result in different voltage levels within the power traces. Next a correlation is attempted to be found between the hypothetical values, obtained with the Hamming weight, and actual power measurements from these different keys[28]. Once such a correlation is found, traces will be used to deduce parts of the key. The benefit of this method compared to DPA is that it generally requires less traces

Side-channel attacks work by gathering a large amount of measurements from the target device. Power measurements in case of a power analysis attack, time for a timing attack, etc. This data is gathered while the device is performing cryptographic operations which then can be used to reveal the key.
The gathered data is then processed through various mathematical and statistical functions to associate the data to the operations the device is performing. Then this data will be tested whether at any point there are significant deviations.
To counter side-channel attacks it is simplest to introduce certain randomization. According to Kocher et al.[15] it is crucial to introduce information unrelated to the important operations. This throws off any timing or power analysis attacks due to it linking with the unpredictable information. This can be done by adding randomized interaction with data, random wait times, or other unpredictable actions. Another method against timing attacks is one with a fixed-time implementation. If every execution of the program takes the exact same amount of time, there is no longer a possible correlation between the time taken and the output of the program.
Side-channel attacks are focused around recovering either the key used for encryption or the internal state of the block cipher with which this may be divulged. Methods for this type of attack would be DPA and CPA which are able to determine the key. In this thesis instead of attacking side-channels, they are analysed. The algorithms are observed to determine whether there are any possible security leaks, and not necessarily attempting to attack these leaks. One of the methods to determine these leaks is with Test Vector Leakage Assessment.

2.2.2 Test Vector Leakage Assessment

For determining whether there is any information leakage occurring within the device executing the cryptographic algorithm, several statistical methods can be used. This thesis applies Test Vector Leakage Assessment. TVLA allows the gathered power traces from the target device to be analysed and used to determine whether the device has power leakage. If the device does have leakage, some methods may also be used to determine whether this leakage is exploitable by attempting to recover the encryption key.

TVLA is a hypothesis testing method. These methods are based around a null hypothesis (H_0) and an alternative hypothesis (H_a). Generally, the null hypothesis is phrased for there to be no statistical significance, and the alternative hypothesis to have statistical significance. After collecting data and using an appropriate statistical test a conclusion can be drawn from the data. This conclusion will either reject the null hypothesis, or fail to reject it. If the results show a significant difference the null hypothesis is rejected.

TVLA contains two different categories of tests[23]: general and specific tests. General tests are used for ascertaining whether there is any leakage in the cryptographic algorithm, while specific tests are for confirming whether the leakage that occurs is also exploitable. This exploitability is not guarenteed as the leaked information may not contain information with which the key can be revealed.

The general test has two main variants, the Fixed-vs.-Random Data and the Fixed-vs.-Random Key test. If the general test shows significant leakage, this may either be leakage that is or is not exploitable. It merely shows the presence of such leakage. This thesis applied the first variant, Fixed-vs.-Random Data. For this variant there are two groups of plaintext, as the name implies, one with fixed plaintext and the other with a randomized plaintext to find significant differences in power consumption. This method uses the same encryption key for all the plaintexts. The other variant would be Fixed-vs.-Random Key, which randomizes the key while keeping the plaintext unchanged.

The specific test uses Random-vs.-Random data. This type of test targets the inner workings of the program that may be exploited to recover the key. If this test fails such that there is a significant difference, it indicates that the leakage may be exploitable and can thus be used to recover the key.

To conduct either of these tests, power traces will need to be gathered. These traces are the power measurements over a span of time captured during the execution of an algorithm. The gathered traces will be split according to which test is being done. In the case of Fixed-vs.-Random data this means that power traces where the fixed plaintext was used is put into one dataset (A), and the power traces with a randomized plaintext are put into another dataset (B).

Having these two datasets allows the application of Welch's t-test, displayed at equation 1, to the datasets, as explained below.
The two datasets, possibly of differing sizes, are compared against eachother to check for statistically significant differences at any point of time within the power trace.

Since each power trace is a vector of measurements across time, the mean power consumption (the power consumption of all samples at that point in time, divided by the amount of samples) and standard deviation (how much the power consumption varies from the mean) of each sample can be represented as vectors over the same points in time. Therefore the Welch's t-test must be applied to every single point in the power trace.
Each t-test is run twice on independent data sets, and if for a given test the same points in time in both datasets exceeds the threshold of 4.5 standard deviations then there is leakage.

$$t = \frac{X_A - X_B}{\sqrt{\frac{{S_A}^2}{N_A} + \frac{{S_B}^2}{N_B}}} \tag{1}$$

Equation 1: Welch's t-test

The gathered traces split into two different datasets are labeled here as A and B. The intermediate values to compute the t-value are X_A, which is the average of all the traces in group A, and X_B, the average of all traces in group B. S_A contains the sample standard deviation of all the traces in group A, and S_B the sample standard deviation of all the traces in group B. N_A and N_B are the amount of traces in each group. Since the Welch's t-test is to be run on each point within the trace, these intermediate values will have to be recalculated for each point, with the exception of N_A and N_B.

2.3 ChipWhisperer

ChipWhisperer devices provide a capture tool for testing algorithms and countermeasures, the full list of tools explained by O'Flynn and Chen [20]. The devices are focused on power analysis attacks, fault attacks, and voltage and clock glitching. The voltage and clock glitching may cause a program to skip key operations such as password checks, while the power analysis attacks exploit patterns in the power consumption of the device. This thesis uses such a ChipWhisperer device for the power analysis of captured traces without fault injection.

2.3.1 Hardware

The ChipWhisperer device is split into two sides; the capture- and the target-side. The capture side of the board captures the power consumption from the target side. This capture side features a customized oscilloscope to be able to measure very small signals during the clock cycles. The device used in this thesis is the ChipWhisperer-Lite 32-bit containing a STM32F3 target board, featuring a 32-bit STM32F303RCT7 Arm Cortex-M4 processor[5]. The R indicates a total of 64 pins on the processor, while the C indicates a total of 256KB flash memory. The T shows that the `LQFP` package (Low-profile, Quad Flat Package mechanical data) is used, and "7" meaning the device is to be used in the temperature ranges of -40 to 105°C
The desired program is run on the target side of the board. For this thesis the target side is executing the various AES implementations that are tested.

[5] https://www.st.com/resource/en/datasheet/stm32f303rc.pdf

For using the ChipWhisperer device there are multiple methods to connect to external devices. The method used in this thesis was connecting the device via USB to a computer which is running a virtual machine. Within this virtual machine jupyter notebooks (explained in the Software section) can be accessed to communicate with the ChipWhisperer. These would include tutorials on how to setup and use the device. Including tutorials on how to capture power traces, flashing the program onto the target device, and various methods for quickly performing TVLA.

The device gathers power measurements by sampling the target device at set intervals. Samples are power measurements taken from the target device which can later be analysed. The higher the amount of samples the more accurate or longer the measurement may be. Such a sample is then stored in the capturing board to be retrieved later. Combining multiple of these samples results in a power trace that is used to performed the analysis. With limited storage for the power trace samples it may be necessary to increase the decimation rate. The decimation rate controls whether a sample is stored or not. Increasing the decimation rate leads to greater intervals between stored samples. This would allow for longer processes to be measured yet this will be at the cost of accuracy.
The ChipWhisperer-Lite 32-bit supports a samplesize of 24573[6]. If samples are required to be taken over a longer amount of time the only option is to increase the decimation rate. The ChipWhisperer-Husky[7] provides a streaming mode for the samples, which would allow the usage of all samples without decimation by sending samples while the program is running. This ensures the buffer where the samples are stored does not fill.

2.3.2 Software

The ChipWhisperer platform features the ability to place triggers around a set of code[20] using the provided functions from `simpleserial.h`. These functions set boundaries for the capture device on where it starts and stops measuring power consumption, allowing precise measurement of specific functions while ignoring others. Functions to communicate with the ChipWhisperer board are included, allowing the sending and receiving of data. After the program has been executed, the power trace can be requested from the capture board which then can be analysed within the Jupyter notebooks.

Jupyter Notebooks allow quick reproduction of specific commands. Being able to easily save and re-run parts of code proves useful for running experiments which may be run multiple times, or with slight variants. The ChipWhisperer contains various pre-made modules for conducting power analysis. It is possible to capture traces with a single command which otherwise would require separately sending and receiving data. Similar tools are provided to quickly calculate t-values for conducting TVLA on the received trace data. The device contains multiple tutorials to get familiar with using Jupyter Notebooks and connecting the device to send your first data back and forth. It guides you through processing the received data to conduct TVLA and interpret its results.

[6] https://rtfm.newae.com/Capture/ChipWhisperer-Lite/
[7] https://rtfm.newae.com/Capture/ChipWhisperer-Husky/

The ChipWhisperer includes a software module to calculate values for the Welch's t-test. This is the python module (`scipy.stats`) version 1.8.1[8], which is used to calculate the values of each point within the datasets gathered for TVLA. The method used in this module is called `ttest_ind(a, b)`, with `(a, b)` being the two datasets to compare. Then, the value is calculated as shown in Equation 1.
The `ttest_ind(a, b)` function has various other functionalities, however, as these were not used during this thesis they are not discussed.

[8] https://github.com/scipy/scipy/blob/v1.8.1/scipy/stats/_stats_py.py#L5944-L6216

3 Related Work

The disadvantage of using the ChipWhisperer board to take the power measurements is that the measurements are restricted by the sampling rate of the oscilloscope. Crocetti et al. [7] suggests an approach to simulate these measurements and thereby removing the main drawbacks of the physical measuring. This simulation works according to a gate-level netlist, an overview with all the logic nodes. This simulation is able to calculate how much power would be used each clock cycle. Being able to increase the gathering speed of samples and not having to have a separate device is beneficial for larger projects. A drawback of using simulations is that the calculations use models which are not exact, and the calculations themself take more time to complete.

The benefit of the ChipWhisperer device is being able to add triggers within the algorithms making it easy to isolate the required power trace. Without this feature, extracting specific parts of the power trace would be more difficult. Trautmann et al.[27] has found a way to isolate the required parts of the power trace when there are no triggers, given that the used algorithm does not include randomized delays. Usually without triggers parts of power traces are isolated by matching the power measurements to known power templates, however, these templates are not always available. Trautmann et al. have found a way to isolate cryptographic operations, given that the algorithm is in constant time. This requirement is the reason why this recognition method does not work with algorithms containing randomized delays.

Much like this thesis, Jayasinghe et al.[12] investigated multiple AES modes for their vulnerabilities against power analysis attacks. They show that all the commonly used modes (ECB, CBC, CTR, OFB, CFB), which were also previously discussed, are vulnerable against Correlation Power Analysis (CPA) attacks for which they used a physical prototyping board. They also noted that, from their suggested rate of change for the keys and initialization vectors, CTR, CFB, and OFB modes are more resistant against power analysis attacks than ECB- and CBC-mode. This thesis also compares the results from ECB, CBC, and CTR-mode and their vulnerability to power analysis attacks.
Petrvalsky et al.[22] further compared the side-channel resistance between different microcontroller cores, both implementing AES. They discuss how many traces are required to reveal the key with a certain confidence interval given two different measurement methods. They discovered the optimal amount of traces to be around 1000 or 2000, depending on the method used to measure the encryption. They also tried different power measurement models, but only tested these on an S-box implementation. The S-box is one of the components of AES, other components may lead to a different optimal amount of traces. Similar to Petrvalsky et al.'s recommended amount of traces, this thesis uses a total of 2000 traces per mode.

According to O'Flynn and Chen[21] typical side-channel attacks on AES focus primarily on AES-128 with ECB mode. They decided to use CPA against AES-256 with CBC mode, which was succesfull after gathering only a hundred traces. After gathering a hundred traces they were able to correctly guess the 13th and 14th roundkey. Knowing the roundkey and its round will make it possible to recalculate the entire key, and so the encryption can be broken. This thesis also compares different modes than the commonly researched ECB mode.

On the topic of hardware attacks, Gnad et al.[11] show a new security threat from boards containing

both analog and digital signals combined. They show that Analog Digital Converters can be a source of leakage, and have succesfully retrieved an AES key with this type of data. This confirms that side-channel attacks can be done with a wide variety of sensory data.
Building further on hardware attacks, Blömer and Seifert[2] present an implementation independent fault attack for AES. This means that no matter how AES is implemented on the device, it would still be attackable. This attack works on the assumption that the attacker is able to control a specific bit. By being able to detect when an invalid ciphertext is created, it is possible to deduce the key by changing bits. When an incorrect ciphertext is created by this forced change of bit, it becomes known what the value should have been at that position (the inverse).
Lastly, Mangard et al.[18] attacked a masked implementation of AES, as it had been theorized that masking would weaken DPA attacks. While from their experiments it is shown that more traces are required than for unmasked implementations, it is still possible to succesfully attack a masked variant.

For the Test Vector Leakage Analysis, Richter[24] used the χ^2-test instead of Welch's T-test. He argues that the main benefit of Pearson's χ^2-test is that this method considers the entire power measurement distribution instead of just a singular moment. It is shown that this method can also aptly be applied for TVLA, as it can sometimes outperform the t-test. This is however not always the case and should therefore not be used as a replacement. They would best be used in conjunction with each other, as one might detect something the other method would not.
Similar to Richter, Wang and Tang[29] investigate the strengths and weaknesses of TVLA. While not comparing it against a different method, they still list shortcomings and possible solutions. Some of these drawbacks include the unreliability of results. If TVLA is done without enough traces, or poor quality ones, it may lead to the conclusion that there is no leakage while there could be. They also note the common issue that while leakage may be detected, it does not have to be exploitable. A device can have leakage and still be perfectly secure, however, this leakage may also become a clue to investigate further.
Roy et al.[25] noticed the same issue of TVLA not being quantifiable. They fixed this by creating a model which allows the TVLA leakage results to be quantified into attack success rate. This gives a moderate guideline on how secure an algorithm is by granting it a value between 0 and 1.

4 Implementations

The implementations used in this project have various unique properties compared to eachother, the general workings of these implementations and their differences are described here.

4.1 Tiny-AES

The first implementation, Tiny-AES[9] is considered as baseline with no particularly unique properties, encrypting only according to how AES is described. The single peculiarity about this implementation being that it contains a LUT-based S-box instead of manually calculating the values to be substituted with. The LUT-based S-box is a lookup table of the substition values which are calculated beforehand, reducing compution time. Tiny-AES has ECB, CBC, and CTR mode implemented, and the pseudocode for this implementation is at Section 8.3.1, and the cipher structure can be seen in Listing 1.

```
1  Cipher(state, RoundKey) {
2      AddRoundKey(0, state, RoundKey);
3      for(round in 1 to Nr) {
4          SubBytes(state)
5          ShiftRows(state)
6          MixColumns(state)
7          AddRoundKey(round, state, RoundKey)
8      }
9      SubBytes(state)
10     ShiftRows(state)
11     AddRoundKey(Nr, state, RoundKey)
12 }
```

Listing 1: Tiny-AES cipher

4.2 Masked-AES

Next, there is an implementation containing a masked version of AES[10], with the pseudocode in Section 8.3.2. This design is based around the masking scheme described by Mangard et al.[17]. Masking is used as a countermeasure against side-channel attacks. These masks are randomized at the start of the encryption of a block. The required inverse for decrypting and other intermediate masks are calculated at the same time. These masks are then applied to the roundkey addition, S-box substitution, and MixColumns. This version also has ECB, CBC, and CTR mode implemented. The cipher structure is visible in Listing 2.

```
1  CipherMasked(state, RoundKey) {
2      InitMasks(mask, RoundKey) {
3          generateRandomMasks() //m1, m2, m3, m4, m, m'
4          calcMixColmask(mask); //Calculate m1',m2',m3',m4'
5          calcSboxMasked(mask); // m' -> m
```

[9] https://github.com/kokke/tiny-AES-c

[10] https://github.com/CENSUS/masked-aes-c/blob/main/aes.c

```
6           maskKey(RoundKey)
7       }
8       remask(state, mask[6-9])
9       AddRoundKey(0, state, RoundKeyMasked);
10      for(round in 1 to Nr) {
11          SubBytesMasked(state)
12          ShiftRows(state)
13          remask(mask[0,1,2,3,5])
14          MixColumns(state)
15          AddRoundKeyMasked(round, state, RoundKeyMasked)
16      }
17      SubBytesMasked(state)
18      ShiftRows(state)
19      AddRoundKeyMasked(Nr, state, RoundKeyMasked)
20  }
```

Listing 2: Masked-AES cipher

4.3 PQClean

Another implementation[11] introduces a bitslicing variant. Here, bits from the internal state that will be encrypted are interleaved. Instead of a lookup table for the S-box the results are manually calculated instead. This S-box is based on the circuit discussed by Boyar and Peralta[3]. This version only has ECB and CTR mode implemented. The pseudocode for this implementation is in Section 8.3.3, and the pseudocode for a single round is visible in Listing 3.

```
1  Bitslice_SubBytes(state) {
2      manually calculate value for each byte
3      y14 = x3^x5
4      y13=x0^x6
5      y9=x0^x3
6      ...
7      state[0]=s7
8  }
9  ShiftRows(state)
10 MixColumns(state)
11 AddRoundKey(round, state, RoundKey)
```

Listing 3: PQClean round

4.4 T-tables

The last implementation uses T-tables[12], precomputed tables for performing the substitution and MixColumns, and other methods such as loop unrolling in order to quickly encrypt the plaintext. This implementation only has the ECB mode implemented. T-tables are calculated by going through all the values in the S-box, and then multiplying them with all 4 columns of the MixColumns matrix, putting them respectively in 4 separate tables. T-Tables save time during execution by no

[11] https://github.com/PQClean/PQClean/blob/master/common/aes.c
[12] https://github.com/pjok1122/AES-Optimization/blob/master/aes.c

longer having to do the calculations during the encryption. The pseudocode for this implementation is in Section 8.3.4.

4.5 Other implementations

Beyond these four implementations, there are other variants. The other considered implementations were Adiantum[13] and one designed for ARM Cortex-M3[14]. The reason these implementations were eventually not chosen is because they were implemented with assembly instructions.

[13] https://github.com/google/adiantum/blob/master/benchmark/src/aes.c
[14] https://github.com/Ko-/aes-armcortexm

5 Methodology

This section focuses on how the experiments are performed. The steps to perform the profiling and the gathering of power consumption measurements are explained and decisions made for these experiments are clarified.

5.1 Profiling

Before taking power consumption measurements, execution time and other metrics are measured for each implementation. These metrics include the size of the compiled programs, important for smaller, embedded, devices that do not have as much storage as typical PCs. Other storage metrics, such as heap and stack usage, are also measured including the size of various large arrays such as the ones used for the S-box lookup table. The execution time is measured in clock cycles, and the memory sizes are measured in bytes.
Each implementation is profiled on two devices. First, the implementations are profiled on a PC, next, the implementations are profiled on the ChipWhisperer board. The PC on which the implementations are run has an Intel Core i7-9750H CPU. The Motherboard is an HP 85FA 42.34. 16GB of RAM is present on the PC. The CPU runs at a clock frequency of 2.6GHz while the ChipWhisperer target board has a clock frequency of 72MHz.

5.1.1 PC

The speed of the various encryption methods for each implementation is measured using the `__rdtsc()` function from `x86intrin.h`[5]. These function calls are placed around key functions such as the encryption call, S-box, and roundkey addition, resulting in accurate performance measurements when averaging these results over multiple thousand iterations. Then, the filesizes of the C-files and executables are measured using the `ls` command in the terminal. Next, the amount of instruction- and data references and cache misses are measured using Valgrind's tool Cachegrind[15]. The cache misses are from first- and last-level (second-level) cache. Lastly, the memory usage is measured using the tool `Memusage`[16]. The memory usage displays how large the stack and heap are for the program. The stack contains data which is automatically deallocated once it is out of scope, and the heap contains data which has to be manually deallocated. This shows how much memory is required to execute the program.
For compiling gcc (Ubuntu 9.4.0-1ubuntu1 20.04.1) 9.4.0 is used. The files are compiled with the command "`gcc -o program aes.c`". Any other files which are required to be included are added to this command.

5.1.2 ChipWhisperer

Since the ChipWhisperer board does not have cache, and has different parts, the general performance differs from the measurements on the local PC. These measurements give a general comparison between the various implementations and how well they perform compared to eachother.

[15] https://valgrind.org/docs/manual/cg-manual.html
[16] https://www.man7.org/linux/man-pages/man1/memusage.1.html

The profiling on the STM32F303RCT7 microcontroller went as followed:
For gathering execution time measurements the value of the cyclecounter register are measured at the start and end of the encryption calls, the same positions as for the other execution time measurements on the local PC.
When programming the target board the total size that is being flashed is shown. This gives the size values for the microcontroller.

5.2 Experimental Setup

For the measuring of power consumption from the various implementations this thesis uses jupyter notebooks. Two experiments are run for each of the implementations and their modes. The first experiment measures the power consumption of encrypting a single block of plaintext, 16 bytes. The second experiment measures the power consumption of encrypting 96 blocks of plaintext, 1536 bytes. This value of 1536 is chosen as it is the closest larger number to the actual frame size of an ethernet packet and still a power of two, allowing to create exactly 96 16-byte blocks of data. The maximum frame size for ethernet is 1526 bytes according to IEEE 802.3 standards[17].

For adding custom code to the ChipWhisperer, the already existing folder `simpleserial-aes` from the directory `hardware/victims/firmware/` is copied. Adding any C-files to that folder allows the compilation to include those files. Next, `simpleserial-aes.c` is edited to call the encryption functions and to accept the corresponding simpleserial read- and put commands.
For running the experiment, various functions are written to improve the ease of use. One function is able to create random plaintexts and also set a fixed plaintext at the start for conducting the Fixed-vs.Random Data test of TVLA. This is required because the function to randomly generate key/text pairs provided by ChipWhisperer does not support plaintext sizes of 1536 bytes. Next, a larger function is made which handles the function calls to the different implementation modes (ECB,CBC,CTR). Simplified function calls allows easy reproduction for gathering the 1000 traces per category.

As the 1536-byte plaintext is too large to transmit immediately towards the target device, it is divided into smaller chunks to be sent. Simpleserial is limited to sending a maximum of 255-bytes[18] as the function for sending data takes an 8-bit value for size. A similar issue occurs with receiving the ciphertext. The 1536-byte text is split into blocks of 64 bytes, resulting in 24 separate transmissions in order to send or receive the full plaintext. For the smaller plaintext this function is edited to only send and receive a single 16-byte block. Splitting the blocks of the larger plaintext into chunks of 16-bytes would increase the time delay. Since every transmission would need to be delayed until the previous one has been sent, it is more beneficial to send larger chunks of text at once. The decided transfer size is 64-bytes as it is a combination of being a multiple of the blocksize and still small enough to not cause transmission issues when sending larger chunks of data per transfer.

After the plaintext is received by the target, the capture board is armed and the target board receives a signal to start encryption. Before the actual encryption starts, various initialization

[17] https://www.ieee802.org/misc-docs/GlobeCom2009/IEEE_802d3_Law.pdf
[18] https://chipwhisperer.readthedocs.io/en/latest/simpleserial.html#simpleserial-put

functions are called such as expanding the key or creating an object out of the plaintext and key. Therefore, the triggers indicating where to start and stop measuring power consumption are positioned around the single encryption function call. These triggers ensure only the power measurements of the encryption itself are included, and ignore any initialisation happening before that.

After gathering the trace data of the two datasets, the t-test function is applied to this data as described in Section 2.2.2. Then, these datapoints are turned into a graph with lines displaying the standard deviation point of 4.5, the cut-off point for TVLA.
Graphs of both the actual power consumption and the TVLA points are saved. The graphs with the actual power consumption are annotated and compared against the TVLA graph.

Due to the length of some of the encryption methods the amount of samples taken is set to 24400 for all of the implementations, as explained in Section 2.3. Most of the implementations when encrypting a single 16-byte plaintext do not require the full 24400 and therefore fit accordingly. The larger plaintexts, however, require more than 24400 samples if sampled at the same rate. This leads to increasing the sample decimation rate in order to gather measurements over the full encryption. A higher decimation rate reduces the accuracy, nonetheless, this is necessary as there is no other method of gathering samples over the full encryption of larger plaintexts with this device. The decimation rate is the same for every implementation's modes when encrypting large plaintexts to minimize differences between graphs.

6 Results & Discussion

In this section the results gathered from the experiments are displayed and discussed as in Section 5.1 and Section 5.2.

6.1 Profiling

First, the area and then the time of each implementation is determined. The area of each implementation is determined by the file- and object sizes, showing how much physical storage is required to store the program and how much memory is required to run the program. The time of each implementation is determined by the amount of cycles one encryption takes.
As can be seen in the results below (Table 2) compiled programs vary between the sizes of 22KB and 31KB. It is convenient that the largest compiled program is not larger than 32KB, as storage size is in powers of 2, the sizes 22KB and 31KB would both fall within the 32KB storage size.
The Te[0-3] tables are the T-Tables used, which are only present for the T-Tables implementation. The Mul-Tables are precalculated for multiplication by 2, 3, 9, 11, 13, 14. These tables are exclusively used for calculating the masks for the MixColumns function. They allow quickly determination of the result when multiplying by those factors (2, 3, 9, 11, 13, 14) in the GF(2^8) which is used in AES encryption and decryption. The Mul, Te[0-3], and SBOX sizes are not exclusive to the PC executable. Both the ARM and PC executable contain these tables, but analysis on the PC is sufficient for the purposes of this thesis.

Measurements	TinyAES	Masked	PQClean	T-Tables
AES.c filesize	19622 bytes	35198 bytes	18009 bytes	54643 bytes
Executable filesize	21984 bytes	30952 bytes	26808 bytes	30800 bytes
SBOX size	256 bytes	256 bytes	×	256 bytes
Inverse-SBOX size	256 bytes	256 bytes	×	256 bytes
Te[0] size	×	×	×	1024 bytes
Te[1] size	×	×	×	1024 bytes
Te[2] size	×	×	×	1024 bytes
Te[3] size	×	×	×	1024 bytes
Mul02 size	×	256 bytes	×	×
Mul03 size	×	256 bytes	×	×
Mul09 size	×	256 bytes	×	×
Mul11 size	×	256 bytes	×	×
Mul13 size	×	256 bytes	×	×
Mul14 size	×	256 bytes	×	×

Cells displaying "×" are not applicable to the measurement, and therefore do not have a result.

Table 2: Area [bytes] - Intel i7 (PC)

For the ARM device the program sizes, visible in Table 3, vary from the results gathered from the local PC. This is the size of the executable on the board to run the program. This executable

contains the code for SimpleSerial, the communication method (explained in Section 2.3.2), too and is not just the AES implementation.

Measurements	TinyAES	Masked	PQClean	T-Tables
Executable size	6863 bytes	16327 bytes	11159 bytes	13271 bytes

Table 3: Area [bytes] - ARM Cortex-M4 (CW)

The memory required to run each implementation also differs. Most of the stack size can be attributed to items such as the S-box and other tables. This can be seen in Table 4.

Memory	TinyAES	Masked	PQClean	T-Tables
Heap total	4096 bytes	4096 bytes	5504 bytes	4096 bytes
Heap peak	4096 bytes	4096 bytes	4800 bytes	4096 bytes
Stack peak	544 bytes	2096 bytes	688 bytes	6256 bytes

Table 4: Area [bytes] - Intel i7 (PC) (heap and stack)

Average encryption times (#cycles)	TinyAES	Masked	PQClean	T-Tables
ECB-Total	4660	8812	4203	315
ECB-SubBytes	101	99	143	×
ECB-ShiftRows	19	19	89	×
ECB-MixColumns	185	181	53	×
CBC-Total	4434	8547	×	×
CBC-SubBytes	92	96	×	×
CBC-ShiftRows	20	18	×	×
CBC-MixColumns	169	172	×	×
CTR-Total	4478	9052	8497	×
CTR-SubBytes	95	98	285	×
CTR-ShiftRows	19	20	175	×
CTR-MixColumns	169	198	109	×

Cells displaying "×" are not applicable to the measurement, and therefore do not have a result.

Table 5: Execution time [#cycles] - Intel i7 (PC)

The performance measurements visible in Table 5 display the amount of cycles each general and sub-operation takes. The T-Table implementation has no S-box, shift, or mix measurements due to the way this program is implemented. Compared to the other implementations, the total encryption time for T-Tables, 315 cycles, is 14.8× faster than the Tiny-AES implementation. This means that the effect of implementing T-tables and unrolling loops for AES can vastly improve performance. The masked implementation is the slowest performer, being 1.9× slower than the baseline Tiny-AES. This means that while masking improves the security, it comes at the cost of performance. There are generally no large disparities between different modes of the same implementation, the exception being PQClean. The CTR-mode of PQClean is 2.0× slower than its ECB mode.

The measurements from the ARM device show similar results in Table 6.

Encryption times (#cycles)	TinyAES	Masked	PQClean	T-Tables
ECB	6851	13066	15119	878
CBC	7035	13077	×	×
CTR	7178	13203	15241	×

Cells displaying "×" are not applicable to the measurement, and therefore do not have a result.

Table 6: Execution time [#cycles] - ARM Cortex-M4 (CW)

A similar ratio of performance difference is visible in this data. The masked implementation is 1.9× slower than Tiny-AES, and the T-tables implementation is again 14.8× faster than Tiny-AES. The measurements on the ChipWhisperer board show that the ECB-mode of PQClean is as slow as it's CTR-mode counterpart, contrary to the measurements on the local PC.
The Valgrind data from Table 7 supports the claims made with Table 5 by showing how many instruction references each implementation uses on average per encryption. More information about Valgrind is in Section 5.1.1. As with the cycle performance, there are 9.7× less instruction references for the T-Tables compared to the others, partially explaining the speed difference between the implementations. Similarly, the masked implementation has 1.8× the amount of instructions compared to Tiny-AES, leading to an equivalent increase in amount of cycles. Due to the results being averages, the amount of misses can be a non-integer value.

Average references (#references, #misses)	TinyAES	Masked	PQClean	T-Tables
Instruction references	21517	39084	23717	2221
I1 misses	0.0084	0.0087	0.014	0.0026
LLi misses	0.0081	0.0085	0.013	0.0026
Data references	10735	19790	14011	678
D1 misses	0.032	0.032	0.048	0.010
LLd misses	0.022	0.022	0.033	0.007

Table 7: Valgrind - Cachegrind - Intel i7 (PC)

6.2 Power analysis results

In this section the power- and leakage measurements of the implementations are individually discussed. Graphs without coloured annotations are present in Section 9.1, and the decimation rates of the various implementations are in Section 8.2.

Tiny-AES - ECB

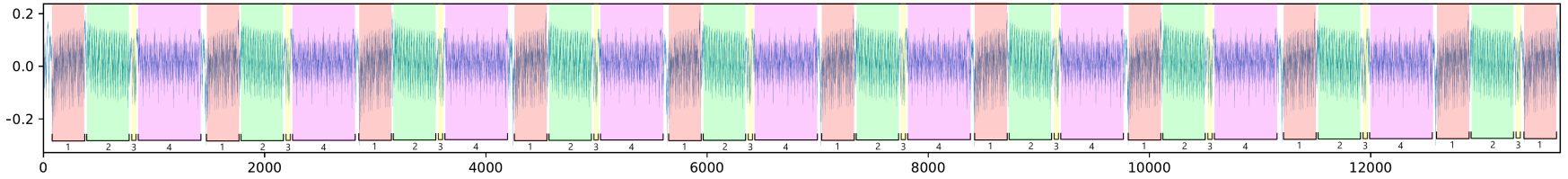


Figure 16: Tiny-AES - ECB - single trace

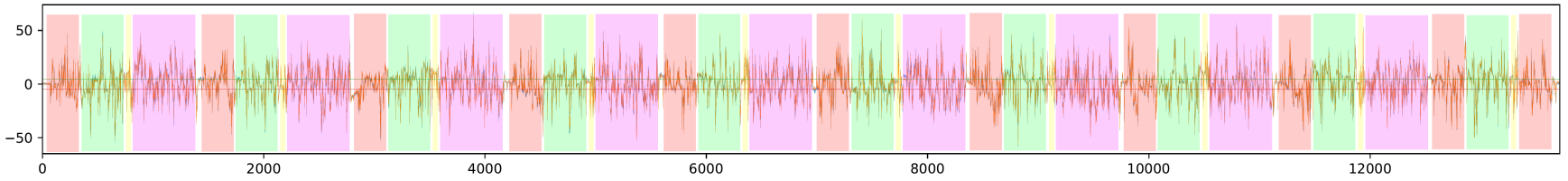


Figure 17: Tiny-AES - ECB - TVLA

Annotations:

1=Add round key 2=Substitution box 3=Shift rows 4=Mix columns

The traces for Tiny-AES in ECB mode are shown in Figure 16 and 17. The graph with power measurements, Figure 16, has been annotated, split up with each key operation of the encryption (SubBytes, ShiftRows, MixColumns, AddRoundKey). Figure 17 visualizes the standard deviation within each point. The red and green lines near the middle of Figure 17 show the cut-off point of 4.5 standard deviations according to the TVLA specification.
As can be seen from the spikes going past these cut-off points, there is leakage throughout the encryption. There are no specific operations which showed an abnormal amount of leakage compared to any of the other encryption operations.

Tiny-AES - CBC

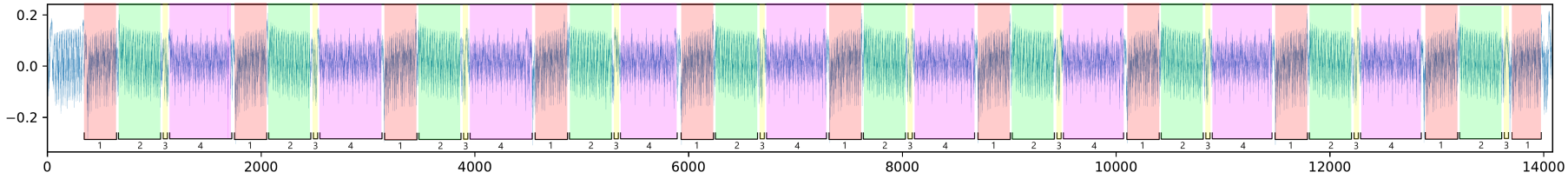


Figure 18: Tiny-AES - CBC - single trace

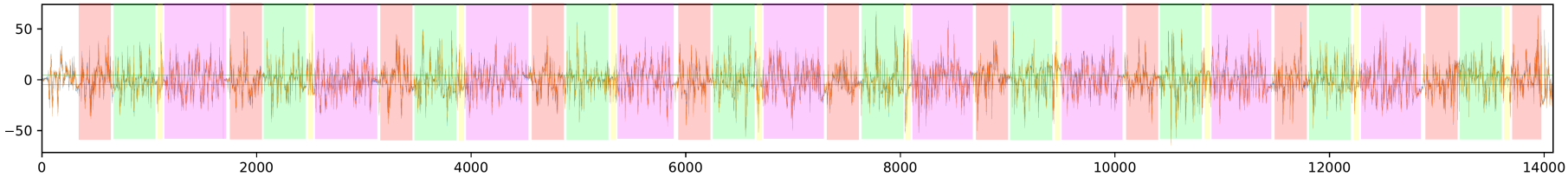


Figure 19: Tiny-AES - CBC - TVLA

Annotations:

1=Add round key 2=Substitution box 3=Shift rows 4=Mix columns

The traces for Tiny-AES in CBC mode are shown in Figure 18 and 19. Figure 18 looks quite similar to the graph for ECB-mode. There is leakage throughout the graph.

Tiny-AES - CTR

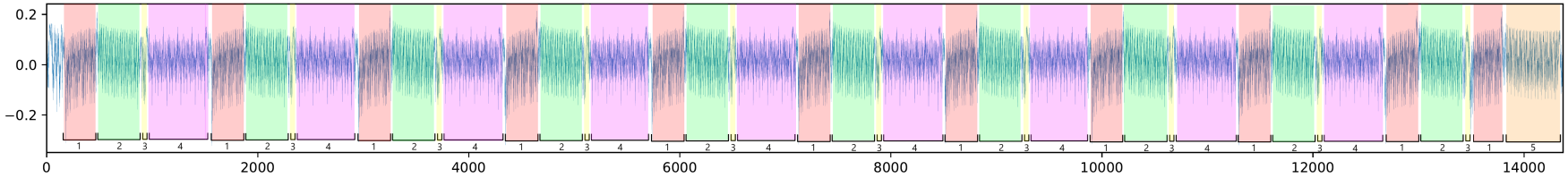


Figure 20: Tiny-AES - CTR - single trace

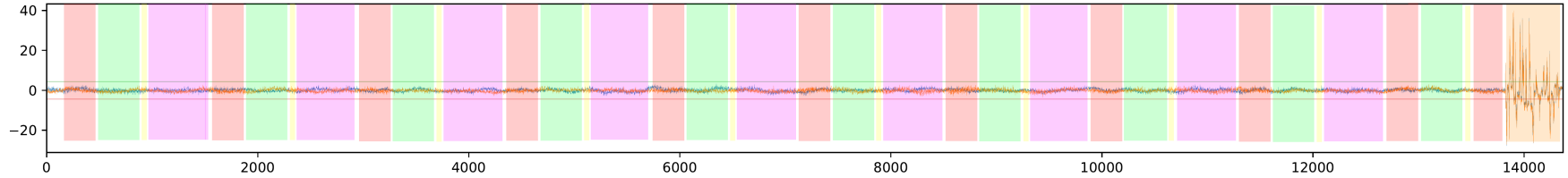


Figure 21: Tiny-AES - CTR - TVLA

Annotations:

1=Add round key 2=Substitution box 3=Shift rows 4=Mix columns 5=Increment IV & XOR plaintext

The traces for Tiny-AES in CTR mode are shown in Figure 20 and 21. Interestingly, compared to the ECB and CBC modes, the Counter mode leads to no leakage being detected with Fixed-vs.Random Data TVLA, except at the very end. From the Figure 20 it can be seen that this leakage occurs when the keystream is XORd with the plaintext, the OTP. As the Fixed-vs.Random Data testing method checks for a dependency between the plaintext and power consumption, there is a peak when the plaintext is used at the OTP, and not at another time in the encryption.

Masked-AES - ECB

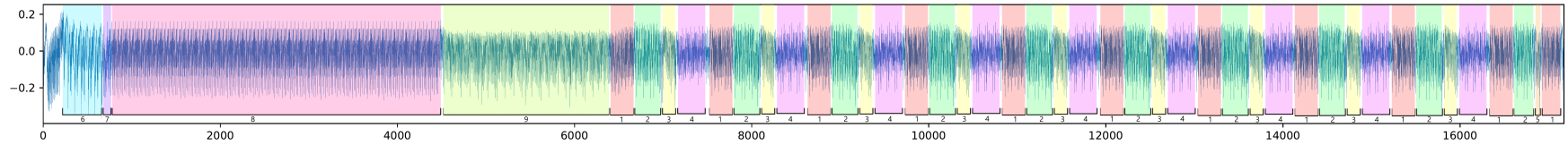


Figure 22: Masked-AES - ECB - single trace

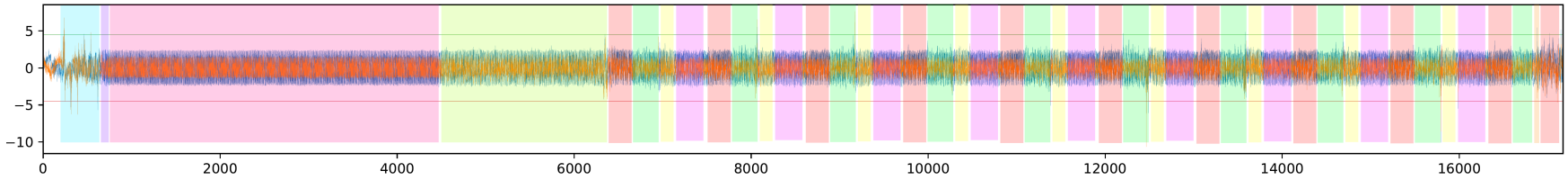


Figure 23: Masked-AES - ECB - TVLA

Annotations:

1=Add round key 2=Substitution box 3=Shift rows and remask 4=Mix columns 5=Shift

6=Randomly generate masks 7=Calculate MixColumn masks 8=Calculate Sbox masks 9=Mask change from M1' M2' M3' M4' to M

The traces for Masked-AES in ECB mode are shown in Figure 22 and 23. For the masked implementation of AES, the ECB mode has little leakage as seen in Figure 23. Except for tiny spikes during the substitution of bytes, there is no leakage detected. Compared to the Tiny-AES implementation, the masked implementation shows less leakage.

Masked-AES - CBC

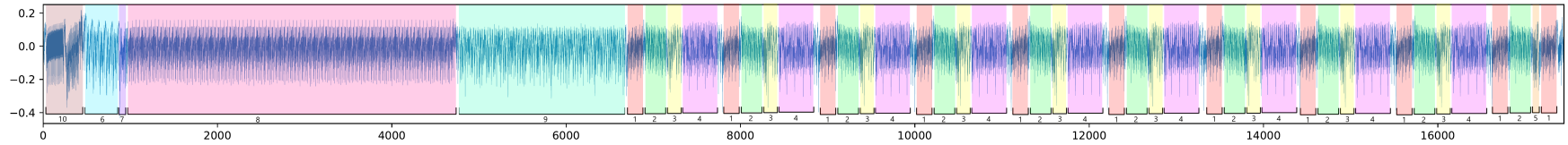


Figure 24: Masked-AES - CBC - TVLA

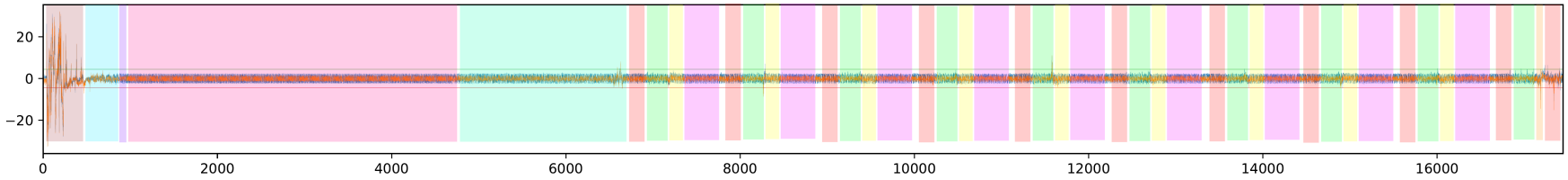


Figure 25: Masked-AES - CBC - TVLA

Annotations:

1=Add round key 2=Substitution box 3=Shift rows and remask 4=Mix columns 5=Shift

6=Randomly generate masks 7=Calculate MixColumn masks 8=Calculate Sbox masks 9=Mask change from M1' M2' M3' M4' to M

10=XOR with IV & Memory copies

The traces for Masked-AES in CBC mode are shown in Figure 24 and 25. For the CBC mode there is a large spike of TVLA leakage at the beginning of Figure 25, but this spike happens before any of the encryption rounds. Just like the ECB mode, there are only a few spikes of leakage happening at section 3 where the shifting and remasking occurs.

Masked-AES - CTR

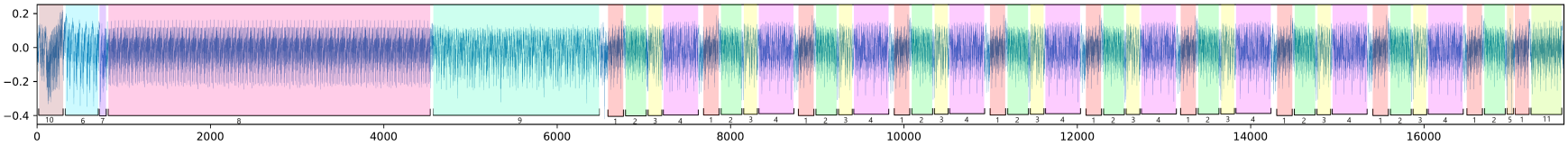


Figure 26: Masked-AES - CTR - single trace

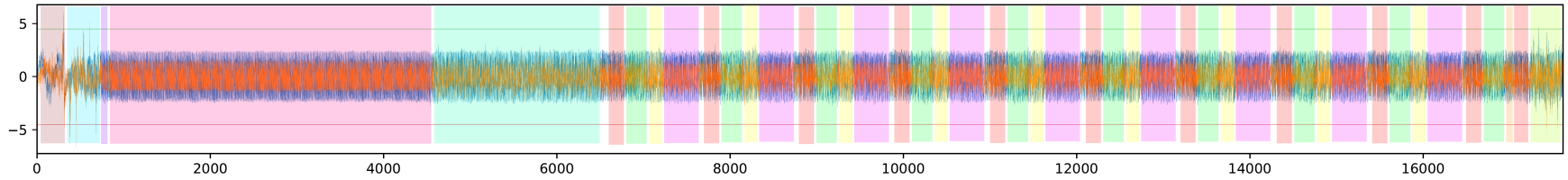


Figure 27: Masked-AES - CTR - TVLA

Annotations:

1=Add round key 2=Substitution box 3=Shift rows and remask 4=Mix columns 5=Shift

6=Randomly generate masks 7=Calculate MixColumn masks 8=Calculate Sbox masks 9=Mask change from M1' M2' M3' M4' to M

10=XOR with IV & Memory copies 11=Increment IV & XOR with plaintext

The traces for Masked-AES in CTR mode are shown in Figure 26 and 27. The Counter mode of the masked implementation only has spikes surpassing the 4.5 lines at the very start and the very end of Figure 27. Within the cipher block section there is no TVLA leakage detected, similar to the counter mode of the Tiny-AES implementation. Again, the small amount of leakage is limited to the OTP.

PQClean - ECB

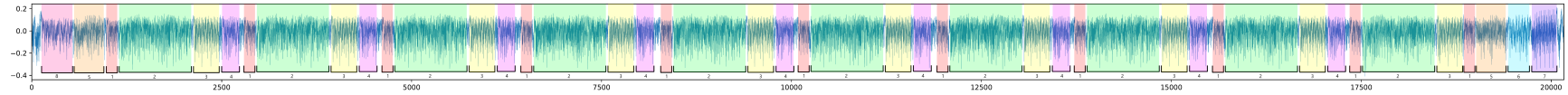


Figure 28: PQClean - ECB - single trace

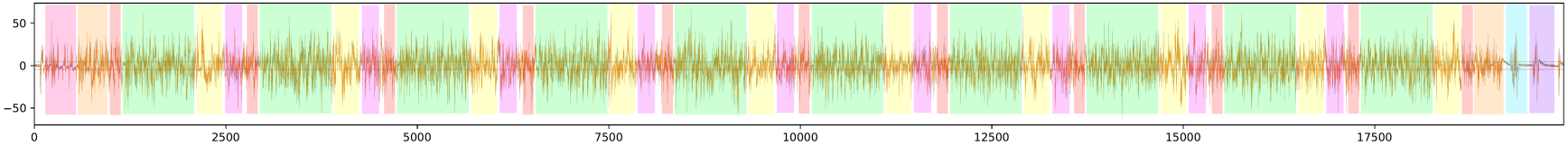


Figure 29: PQClean - ECB - TVLA

Annotations:

1=Add round key 2=Sliced substitution box 3=Shift rows 4=Mix columns 5=Create orthogonality 6=Interleave-out plaintext 7=Move to output 8=Interleave-in plaintext

The traces for PQClean, using bitslicing, in ECB mode are shown in Figure 28 and 29. Unlike the masked variant the PQClean implementation shows no clear improvement on reducing any form of leakage when inspecting Figure 29. The resulting graph shows TVLA leakage throughout the encryption, just as the Tiny-AES implementation.

PQClean - CTR

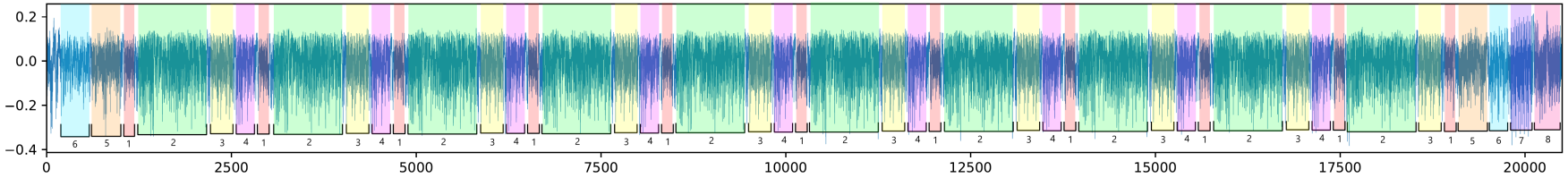


Figure 30: PQClean - CTR - single trace

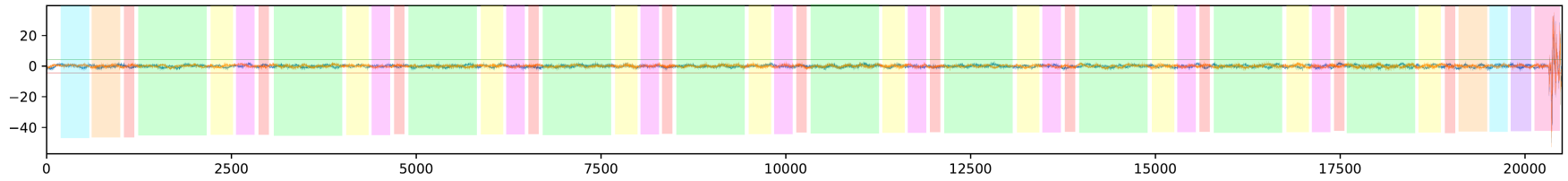


Figure 31: PQClean - CTR - TVLA

Annotations:

1=Add round key 2=Sliced substitution box 3=Shift rows 4=Mix columns 5=Create orthogonality 6=Interleave-out plaintext 7=Move to output 8=Interleave-in plaintext

Much like with the other implementations, PQClean's counter mode, displayed in Figure 30 and 31 shows much less TVLA leakage except a spike at the end of figure 31. This most likely has to do with counter mode only interacting with the plaintext to XOR with the ciphertext at the end, just as the other implementations in counter mode.

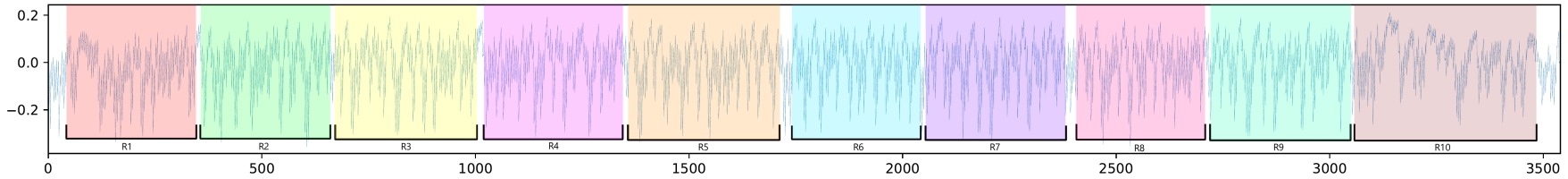


Figure 32: T-Tables - ECB - single trace

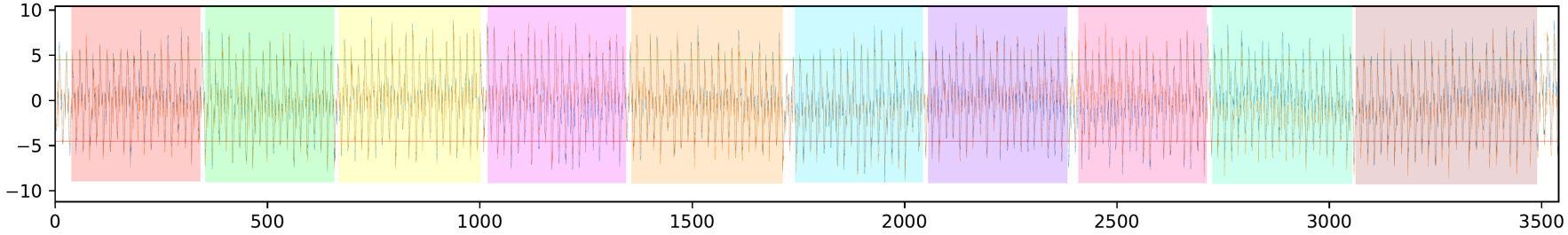


Figure 33: T-Tables - ECB - TVLA

Annotations:

R1-R10 denote the round number. Due to the way T-tables are implemented it is not possible to separate the measurements in the same manner as the other implementations.

The traces for the T-tables implementation in ECB mode are shown in Figure 32 and 33. While the T-table implementation does not show spikes from the TVLA graph as large as other implementations in Figure 33, this does not mean the leakage would be less significant.

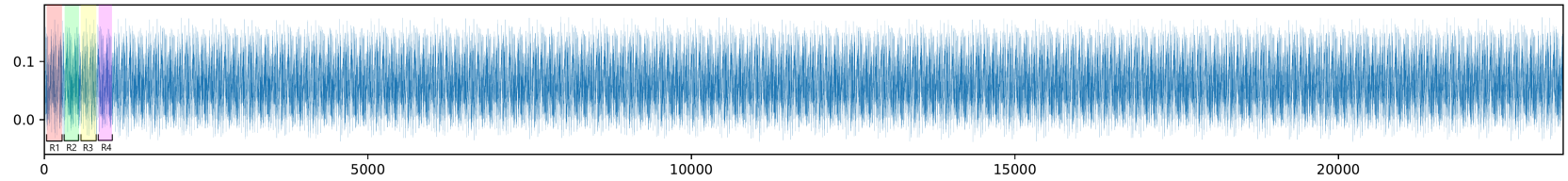


Figure 34: Tiny-AES - ECB - single trace - 1536 bytes

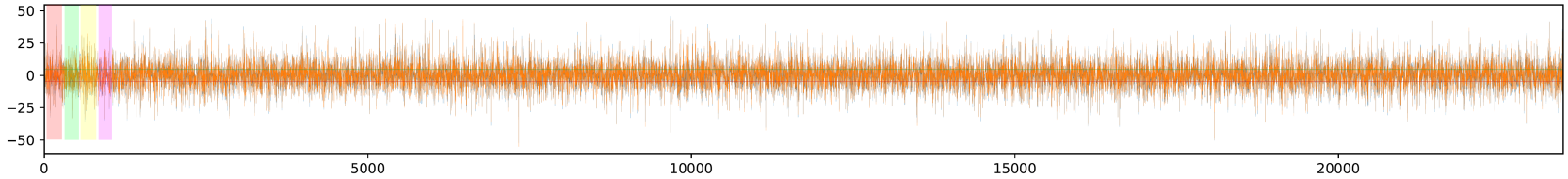


Figure 35: Tiny-AES - ECB - TVLA - 1536 bytes

Annotations:

R1-R4 are the new plaintext blocks. This scales all the way through to R96.

The traces for the Tiny-AES implementation in ECB mode are shown in Figure 34 and 35.
Processing the large 1536-byte plaintext seems to have made no significant changes to the measurements compared to the single block of plaintext in Figure 16 and 17.

Large plaintext: Tiny-AES - CBC

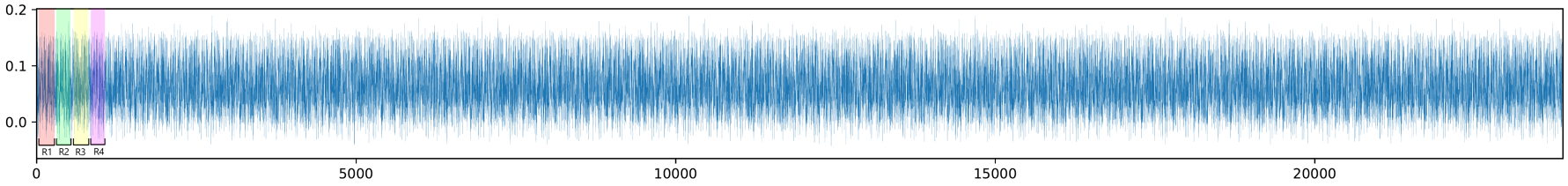


Figure 36: Tiny-AES - CBC - single trace - 1536 bytes

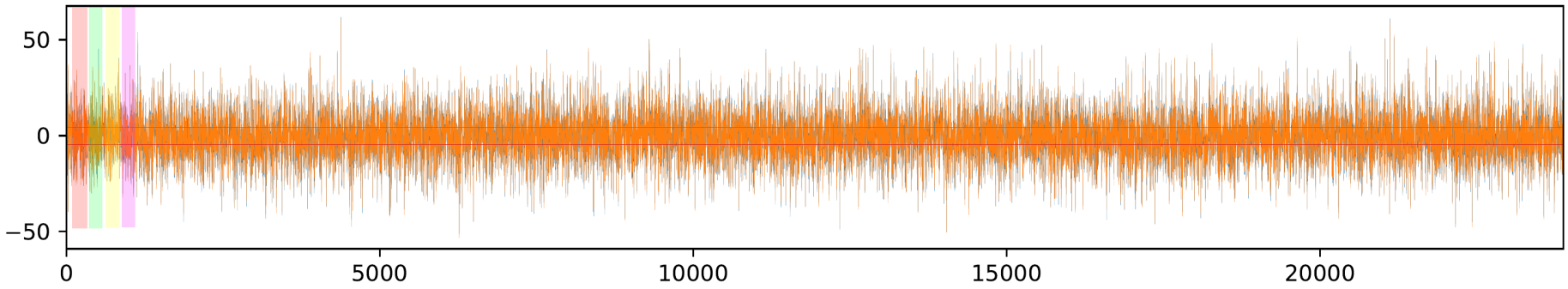


Figure 37: Tiny-AES - CBC - TVLA - 1536 bytes

Annotations:

R1-R4 are the new plaintext blocks. This scales all the way through to R96.

The traces for the Tiny-AES implementation in CBC mode are shown in Figure 36 and 37.
While again no large changes can be seen from the TVLA graph in Figure 37, it can be seen that the individual power graph, Figure 36 is now clearly different from the ECB-mode graph in Figure 34. These differences can be attributed to the cipher-block-chaining.

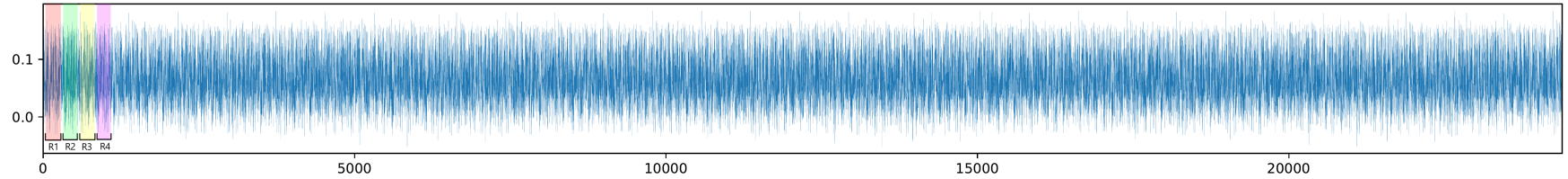


Figure 38: Tiny-AES - CTR - single trace - 1536 bytes

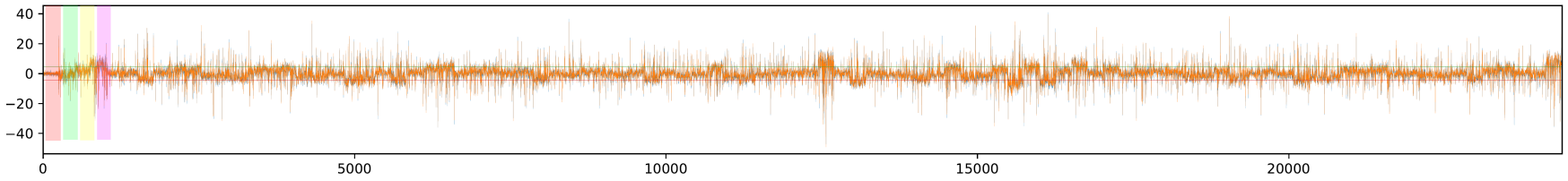


Figure 39: Tiny-AES - CTR - TVLA - 1536 bytes

Annotations:

R1-R4 are the new plaintext blocks. This scales all the way through to R96.

The traces for the Tiny-AES implementation in CTR mode are shown in Figure 38 and 39.
It can be seen that while the TVLA graph still shows leakage, the spikes are significantly smaller compared to the other two modes.

Large plaintext: Masked-AES - ECB

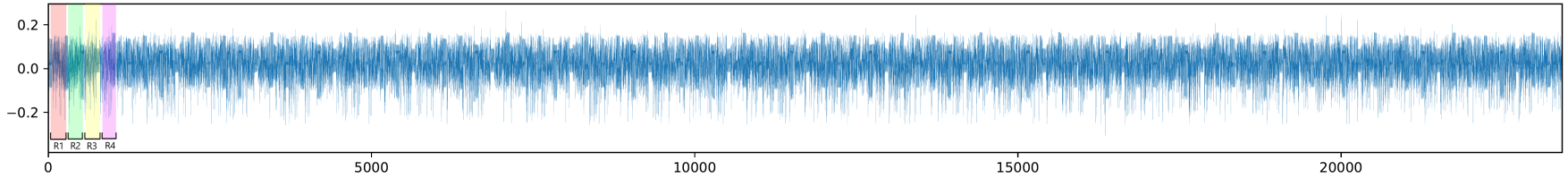


Figure 40: Masked-AES - ECB - single trace - 1536 bytes

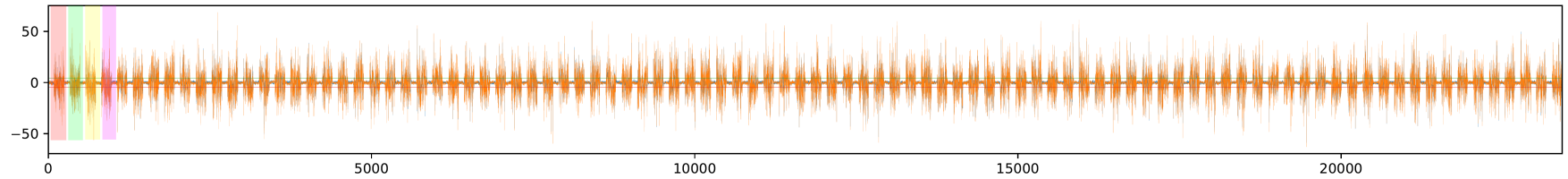


Figure 41: Masked-AES - ECB - TVLA - 1536 bytes

Annotations:

R1-R4 are the new plaintext blocks. This scales all the way through to R96.

The traces for the Masked-AES implementation in CTR mode are shown in Figure 40 and 41.
While the masked variant of the smaller plaintext barely had any TVLA leakage as was shown in Figure 23, when processing the larger plaintext it can be seen in Figure 41 that there is a significant amount of leakage. The TVLA graph does have small gaps which aligns with every separate round, which is when the implementation is generating new random masks

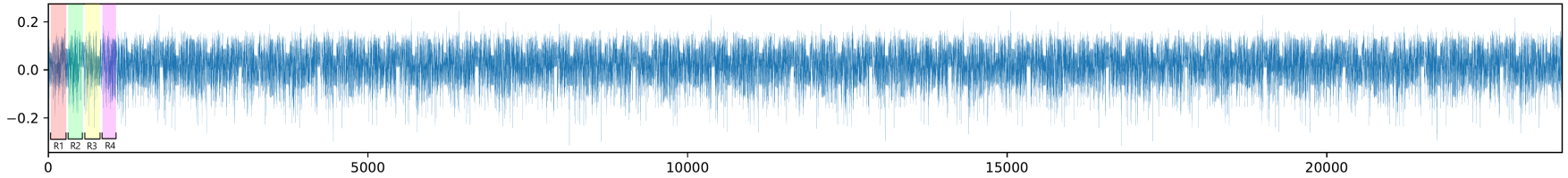


Figure 42: Masked-AES - CBC - single trace - 1536 bytes

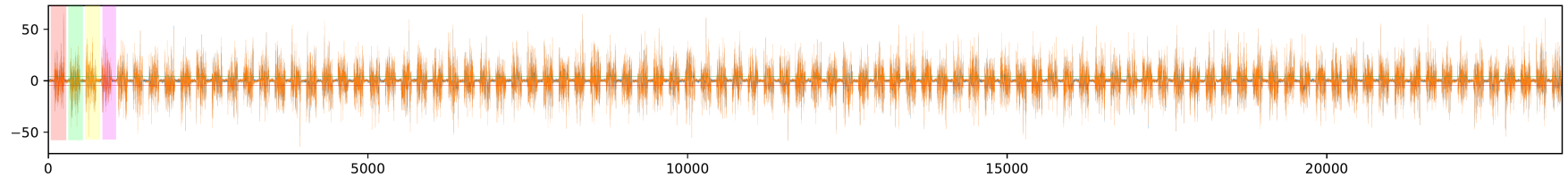


Figure 43: Masked-AES - CBC - TVLA - 1536 bytes

Annotations:

R1-R4 are the new plaintext blocks. This scales all the way through to R96.

The traces for the Masked-AES implementation in CBC mode are shown in Figure 42 and 43.
Similar to the ECB variant, there is now in Figure 43 an increase in leakage visible from the implementation.

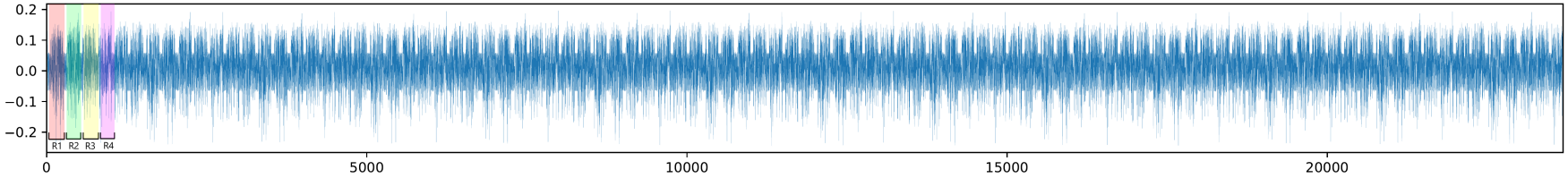


Figure 44: Masked-AES - CTR - single trace - 1536 bytes

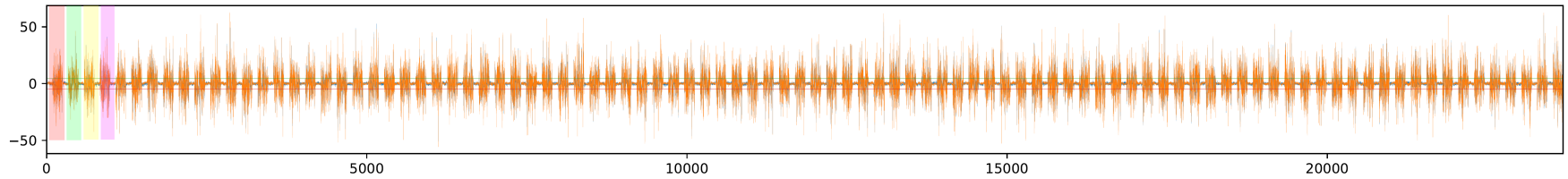


Figure 45: Masked-AES - CTR - TVLA - 1536 bytes

Annotations:

R1-R4 are the new plaintext blocks. This scales all the way through to R96.

The traces for the Masked-AES implementation in CTR mode are shown in Figure 44 and 45.
Just like the masked ECB and CBC mode, it seems that masking is not enough to fully reduce the TVLA leakage when processing larger plaintexts.

Large plaintext: PQClean - ECB

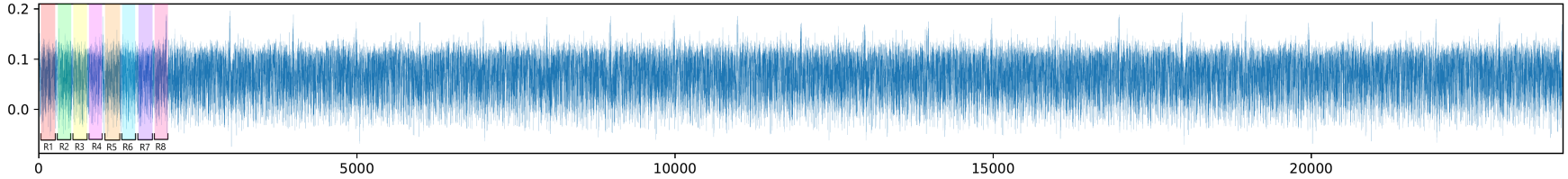


Figure 46: PQClean - ECB - single trace - 1536 bytes

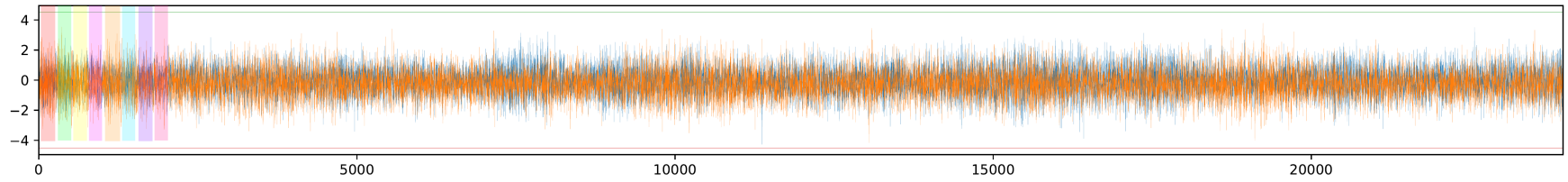


Figure 47: PQClean - ECB - TVLA - 1536 bytes

Annotations:

R1-R8 are the new plaintext blocks. This scales all the way through to R96.

The traces for the PQClean implementation in ECB mode are shown in Figure 46 and 47.
As can be seen in Figure 47, this implementation does not show TVLA leakage when encrypting the 1536-byte plaintext. What is interesting is that the implementation did have TVLA leakage when processing a single block of plaintext.

Large plaintext: PQClean - CTR

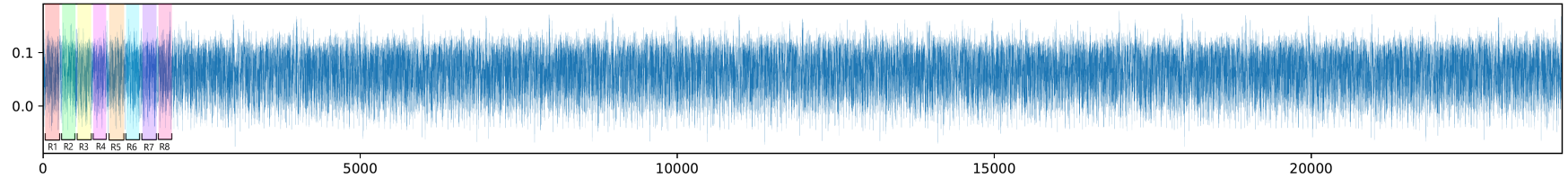


Figure 48: PQClean - CTR - single trace - 1536 bytes

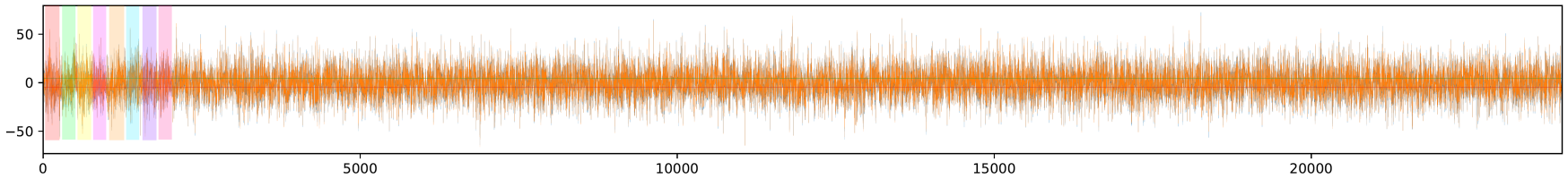


Figure 49: PQClean - CTR - TVLA - 1536 bytes

Annotations:

R1-R8 are the new plaintext blocks. This scales all the way through to R96.

The traces for the PQClean implementation in CTR mode are shown in Figure 48 and 49.
Unlike other implementations, PQClean seems to get worse TVLA leakage results with counter mode compared to the other modes. While the ECB mode displays no leakage, now with counter mode the bitslicing implementation shows leakage throughout the entire long encryption.

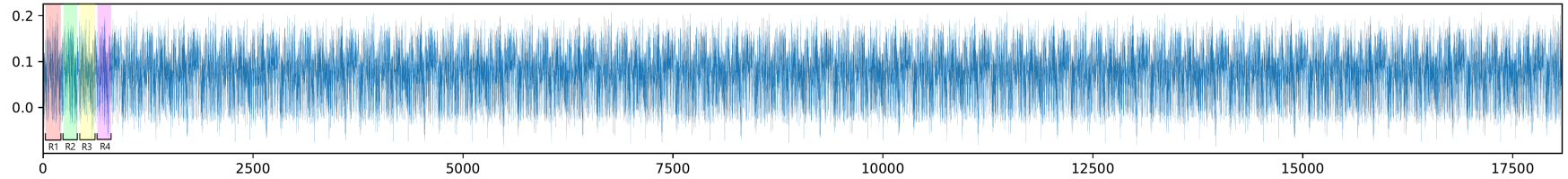


Figure 50: T-Tables - ECB - single trace - 1536 bytes

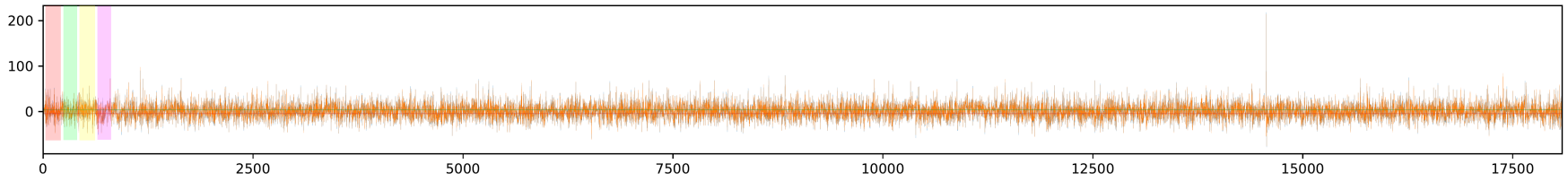


Figure 51: T-Tables - ECB - TVLA - 1536 bytes

Annotations:

R1-R4 are the new plaintext blocks. This scales all the way through to R96.

The traces for the T-Tables implementation in ECB mode are shown in Figure 50 and 51.
This implementation shows a very large amount of leakage. This is not unexpected when taking the results with the smaller plaintext, from Figure 33, into account. Unlike the PQClean implementation, the T-Tables implementation does not behave differently with larger plaintexts, leading to similar results. Interestingly enough there is a very large spike near 15000 samples, though this may be due to the moving of plaintext being included in the trigger-zone.

6.2.1 Summarized Findings

Counter mode is able to isolate TVLA leakage to the OTP instead of over the entire cipher for the 16-byte plaintexts. It is not able to isolate this leakage when encrypting the 1536-byte plaintext.
Tiny-AES consistently displays leakage throughout the implementations, with the exception for counter mode which only shows leakage at the OTP. For the larger plaintext the counter mode shows less leakage than the other modes, but there is still leakage throughout the entire graph.
Masked-AES is able to significantly reduce the detected leakage with smaller plaintexts, having only a few spikes of leakage. With larger plaintexts Masked-AES shows larger spikes of leakage at encrypting, happening every 16-byte block in the 1536-byte plaintext. The large peaks visible at the initialisation of the 16-byte plaintext can be disregarded as there is no interaction with the plaintext or key. This leakage may be due to the random number generator used for the masking scheme.
PQClean shows similar results as Tiny-AES when encryption the 16-byte plaintext, displaying leakage throughout the encryption. When encrypting the larger plaintext in ECB-mode no TVLA leakage is detected, this is not the case with CTR-mode where TVLA leakage is detected throughout the encryption.
The T-Tables implementation consistently displays leakage with both the small and larger plaintext.
Jayasinghe et al.[12] also investigate various AES modes and check their vulnerability. Contrary to this thesis, they investigate a single implementation and attack it with CPA instead of comparing various implementations and the detected leakage. The difficulty of attack allows them to give a resistance rating to each mode of operation, similar to how in this thesis the TVLA graphs are compared with eachother.

7 Conclusion

With security and privacy becoming more valued due to the increasing level of cybersecurity threats, encryption becomes increasingly important. This thesis aims to compare various AES implementations in order to check the presence of side-channel leakage. Exploiting this leakage, which is indirect information leaked during the execution of a program, allows the discovery of items such as the encryption key.
In this thesis the following AES implementations were used: Tiny-AES, Masked-AES, PQClean, and T-Tables. These implementations were profiled on execution time and area (in bytes) before power measurements were taken. Memory usage, filesizes, and cycle count during encryption are some of these metrics. This profiling was done on a PC and a ChipWhisperer board. After profiling, power measurements were gathered from the various AES implementations during encryption with the ChipWhisperer board. This encryption was done over two different sizes of plaintext. One 16-byte plaintext and a 1536-byte plaintext. After power measurements from the encryption are gathered, TVLA was then applied to this data to detect leakage.

When comparing the implementations against Tiny-AES, the Masked- and PQClean implementation performed the encryption slower while the T-Tables implementation performed the encryption fastest. These implementations having special properties for encryption compared to Tiny-AES such as Masking, Bitslicing, and T-Tables lead to those executables being larger than the executable for Tiny-AES. As these executables are flashed on microcontrollers which have limited size, it is important for these executables to not be too large.

Power leakage in AES implementations can be reduced by masking, and there is less detected Fixed-vs.Random Data TVLA leakage when using either bitslicing or the counter mode of operation. As power analysis is used to detect leakage, adding these properties to implementations can lead to a more secure program. While bitslicing is not a countermeasure to side-channel attacks, the measurements show that it can in specific cases reduce the amount of detected TVLA leakage when analysing plaintexts. Tiny-AES consistently displays leakage throughout the implementations, with the exception for counter mode which only shows TVLA leakage at the OTP. For the larger plaintext the counter mode shows less leakage than the other modes, but does consistently leak. Masked-AES is able to significantly reduce the detected leakage with smaller plaintexts, having only a few spikes of leakage. With larger plaintexts Masked-AES shows more leakage. PQClean shows similar results as Tiny-AES when encryption the 16-byte plaintext, but ECB-mode with the larger plaintext is able to reduce the Fixed-vs.Random Data TVLA leakage significantly. The T-Tables implementation consistently displays leakage with both the small and larger plaintext. While the used masking implementation does show leakage when processing larger plaintexts, it could still be secure due to this leakage not necessarily being exploitable. A large difference can be seen when comparing ECB and CBC mode with CTR mode. The CTR mode is able to significantly reduce the detected TVLA leakage as the encryption process itself only uses the plaintext at the very end to XOR with the ciphertext, being a OTP.
CTR mode being able to reduce Fixed-vs.Random Data TVLA leakage and deemed more secure aligns with the findings of Jayasinghe et al.[12].

For future research the other testing methods of TVLA could be applied to these implementations

to compare results. It would be similarly interesting to conduct actual attacks against these implementations, especially the masked variant due to it showing separated spikes of leakage. A different way of generating random numbers for the masking scheme can also be tried, due to the current method displaying leakage. Next would be investigating the behaviour when for example inserting a zero-plaintext or other synthetic messages. Due to the used implementations not being made specifically for the used ARM device, using different implementations or different target devices could lead to other results as leakage is device specific.

References

[1] S. Arrag. Design and implementation a different architectures of mixcolumn in fpga. *International Journal of VLSI Design & Communication Systems*, 3:11–22, 08 2012.

[2] J. Blömer and J.-P. Seifert. *Fault Based Cryptanalysis of the Advanced Encryption Standard (AES)*, page 162–181. Springer Berlin Heidelberg, 2003.

[3] J. Boyar and R. Peralta. *A New Combinational Logic Minimization Technique with Applications to Cryptology*. Springer, Berlin, Heidelberg, 1 2010.

[4] E. Brier, C. Clavier, and F. Olivier. *Correlation Power Analysis with a Leakage Model*. Springer Berlin Heidelberg, 1 2004.

[5] Clang. clang: lib/Headers/ia32intrin.h File Reference. https://clang.llvm.org/doxygen//ia32intrin_8h.html#a2f5de17fb08fef627ccf2103643a82e5.

[6] Council of the European Union. Council conclusions on the future of cybersecurity: implement and protect together. Brussels, May 2024. 10133/24. https://data.consilium.europa.eu/doc/document/ST-10133-2024-INIT/en/pdf.

[7] L. Crocetti, L. Baldanzi, M. Bertolucci, L. Sarti, B. Carnevale, and L. Fanucci. A simulated approach to evaluate side-channel attack countermeasures for the advanced encryption standard. *Integration*, 68:80–86, Sept. 2019.

[8] J. Daemen and V. Rijmen. Aes proposal: rijndael. *NIST*, 10 1999.

[9] M. J. Dworkin. Recommendation for block cipher modes of operation :. Technical report, National Institute of Standards and Technology, 1 2001.

[10] M. J. Dworkin. Advanced Encryption Standard (AES). Technical report, National Institute of Standards and Technology, 5 2023.

[11] D. R. E. Gnad, J. Krautter, and M. B. Tahoori. Leaky noise: New side-channel attack vectors in mixed-signal iot devices. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, page 305–339, May 2019.

[12] D. Jayasinghe, R. Ragel, J. A. Ambrose, A. Ignjatovic, and S. Parameswaran. Advanced modes in aes: Are they safe from power analysis based side channel attacks? In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 173–180, 2014.

[13] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 388–397, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[14] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, Mar. 2011.

[15] P. C. Kocher, J. M. Jaffe, B. C. Jun, and C. R. Inc. US6327661B1 - Using unpredictable information to minimize leakage from smartcards and other cryptosystems - Google Patents, 6 1998.

[16] T. Lugrin. *One-Time Pad*, pages 3–6. Springer Nature Switzerland, Cham, 2023.

[17] S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks*. Springer New York, NY, 1 2007.

[18] S. Mangard, N. Pramstaller, and E. Oswald. *Successfully Attacking Masked AES Hardware Implementations*, page 157–171. Springer Berlin Heidelberg, 2005.

[19] A. Matthews. test vector leakmartcards. *Network Security*, 2006(12):18–20, 2006.

[20] C. O'Flynn and Z. D. Chen. ChipWhisperer: An open-source platform for hardware embedded security research. Cryptology ePrint Archive, Paper 2014/204, 2014. https://eprint.iacr.org/2014/204.

[21] C. O'Flynn and Z. David Chen. Side channel power analysis of an aes-256 bootloader. In *2015 IEEE 28th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 750–755, 2015.

[22] M. Petrvalsky, M. Drutarovsky, and M. Varchola. Differential power analysis of advanced encryption standard on accelerated 8051 processor. In *2013 23rd International Conference Radioelektronika (RADIOELEKTRONIKA)*, pages 334–339, 2013.

[23] Rambus. Test Vector Leakage Assessment (TVLA) Derived Test Requirements (DTR) with AES, 3 2019.

[24] B. Richter. *Advanced side-channel measurement and testing*. PhD thesis, Ruhr-Universität Bochum, 2021.

[25] D. B. Roy, S. Bhasin, S. Guilley, A. Heuser, S. Patranabis, and D. Mukhopadhyay. Leak me if you can: Does TVLA reveal success rate? Cryptology ePrint Archive, Paper 2016/1152, 2016. https://eprint.iacr.org/2016/1152.

[26] W. Schindler. A timing attack against rsa with the chinese remainder theorem. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2000*, pages 109–124, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[27] J. Trautmann, A. Beckers, L. Wouters, S. Wildermann, I. Verbauwhede, and J. Teich. Semi-automatic locating of cryptographic operations in side-channel traces. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, page 345–366, Nov. 2021.

[28] E. Tromer. Power analysis, correlation power analysis. https://cs-people.bu.edu/tromer/istvr1516-files/lecture3-power-analysis.pdf, 2015-2016. [Accessed 1 Jul, 2024].

[29] Y. Wang and M. Tang. A survey of Side-Channel Leakage assessment. *Electronics*, 12(16):3461, 8 2023.

8 Appendix A

8.1 Abbreviations

AES	Advanced Encryption Standard
CBC	Cipher Block Chaining
CFB	Cipher FeedBack
CPA	Correlation Power Analysis
CTR	Counter
DPA	Differential Power Analysis
ECB	Electronic Code Book
EUCO	European Council
IEEE	Institute of Electrical and Electronics Engineers
LUT	Look-Up Table
LQFP	Low-profile, Quad Flat Package mechanical data
MAC	Message Authentication Code
NIST	National Insitute of Standards and Technology
OFB	Output FeedBack
OTP	One-Time Pad
SPA	Simple Power Analysis
SPN	Substitution-Permutation Network
TVLA	Test Vector Leakage Assessment
XOR	Exclusive OR, modulo 2 addition

Table 8: List of used abbreviations

8.2 Decimation rates power measurement gathering

Decimation rate	TinyAES	Masked	PQClean	T-Tables
ECB-Small	2	3	3	1
CBC-Small	2	3	×	×
CTR-Small	2	3	3	×
ECB-Large	120	210	60	20
CBC-Large	120	210	×	×
CTR-Large	120	210	60	×

Cells displaying "×" are not applicable to the measurement, and therefore do not have a result.

Table 9: Decimation rates

8.3 Pseudocode

8.3.1 Tiny-AES

```
#ECB
KeyExpansion(key);
AES_ECB_encrypt(key, plaintext) {
    Cipher(state, RoundKey) {
        AddRoundKey(0, state, RoundKey);
        for(round in 1 to Nr) {
            SubBytes(state)
            ShiftRows(state)
            MixColumns(state)
            AddRoundKey(round, state, RoundKey)
        }
        SubBytes(state)
        ShiftRows(state)
        AddRoundKey(Nr, state, RoundKey)
    }
}

#CBC
KeyExpansion(key);
AES_CBC_encrypt(key, plaintext, IV) {
    for(block in 0 to length/blocklength) {
        XorWithIv(input, iv);

        Cipher(state, RoundKey) {
            AddRoundKey(0, state, RoundKey);
            for(round in 1 to Nr) {
                SubBytes(state)
                ShiftRows(state)
                MixColumns(state)
                AddRoundKey(round, state, RoundKey)
            }
            SubBytes(state)
            ShiftRows(state)
            AddRoundKey(Nr, state, RoundKey)
        }
        iv = output

        //move to next block of data to encrypt
        input += blocklength
    }
}

#CTR
KeyExpansion(key);
AES_CTR_encrypt(key, plaintext, IV) {
    for(block in 0 to length/blocklength) {

        //cipher encrypts IV
        Cipher(state, RoundKey) {
            AddRoundKey(0, state, RoundKey);
```

```
            for(round in 1 to Nr) {
                SubBytes(state)
                ShiftRows(state)
                MixColumns(state)
                AddRoundKey(round, state, RoundKey)
            }
            SubBytes(state)
            ShiftRows(state)
            AddRoundKey(Nr, state, RoundKey)
        }
        XorWithPT(output, plaintext);
        iv += 1;

        //move to next block of data to encrypt
        plaintext+=blocklength;
    }
}
```

8.3.2 Masked AES

```
#ECB
KeyExpansion(key);
AES_ECB_encrypt(key, plaintext) {
    CipherMasked(state, RoundKey) {
        InitMasks(mask, RoundKey) {
            generateRandomMasks() //m1, m2, m3, m4, m, m'
            calcMixColmask(mask); //Calculate m1',m2',m3',m4'
            calcSboxMasked(mask); // m' -> m
            maskKey(RoundKey)
        }
        remask(state, mask[6-9])
        AddRoundKey(0, state, RoundKeyMasked);
        for(round in 1 to Nr) {
            SubBytesMasked(state)
            ShiftRows(state)
            remask(mask[0,1,2,3,5])
            MixColumns(state)
            AddRoundKeyMasked(round, state, RoundKeyMasked)
        }
        SubBytesMasked(state)
        ShiftRows(state)
        AddRoundKeyMasked(Nr, state, RoundKeyMasked)
    }
}


#CBC
KeyExpansion(key);
AES_ECB_encrypt(key, plaintext, IV) {
    for(block in 0 to length/blocklength) {
        XorWithIv(input, iv);

        CipherMasked(state, RoundKey) {
            InitMasks(mask, RoundKey) {
```

```
                generateRandomMasks() //m1, m2, m3, m4, m, m'
                calcMixColmask(mask); //Calculate m1',m2',m3',m4'
                calcSboxMasked(mask); // m' -> m
                maskKey(RoundKey)
            }
            remask(state, mask[6-9])
            AddRoundKey(0, state, RoundKeyMasked);
            for(round in 1 to Nr) {
                SubBytesMasked(state)
                ShiftRows(state)
                remask(mask[0,1,2,3,5])
                MixColumns(state)
                AddRoundKeyMasked(round, state, RoundKeyMasked)
            }
            SubBytesMasked(state)
            ShiftRows(state)
            AddRoundKeyMasked(Nr, state, RoundKeyMasked)
        }
        iv = output

        //move to next block of data to encrypt
        input += blocklength
    }
}


#CTR
KeyExpansion(key);
AES_ECB_encrypt(key, plaintext, IV) {
    for(block in 0 to length/blocklength) {

        //cipher encrypts IV
        CipherMasked(state, RoundKey) {
            InitMasks(mask, RoundKey) {
                generateRandomMasks() //m1, m2, m3, m4, m, m'
                calcMixColmask(mask); //Calculate m1',m2',m3',m4'
                calcSboxMasked(mask); // m' -> m
                maskKey(RoundKey)
            }
            remask(state, mask[6-9])
            AddRoundKey(0, state, RoundKeyMasked);
            for(round in 1 to Nr) {
                SubBytesMasked(state)
                ShiftRows(state)
                remask(mask[0,1,2,3,5])
                MixColumns(state)
                AddRoundKeyMasked(round, state, RoundKeyMasked)
            }
            SubBytesMasked(state)
            ShiftRows(state)
            AddRoundKeyMasked(Nr, state, RoundKeyMasked)
        }
        XorWithPT(output, plaintext);
        iv += 1;
```

```

        //move to next block of data to encrypt
        plaintext+=blocklength;
    }
}
```

8.3.3 PQClean

```

#ECB
KeyExpansion(key);
AES_ECB_encrypt(key, plaintext) {
    interleafBlocks(plaintext)
    Orthogonality(state)

    Cipher(state, RoundKey) {
        AddRoundKey(0, state, RoundKey);
        for(round in 1 to Nr) {
            Bitslice_SubBytes(state)
                    manually calculate value for each byte
                    y14 = x3^x5
                    y13=x0^x6
                    y9=x0^x3
                    ...
                    state[0]=s7
            ShiftRows(state)
            MixColumns(state)
            AddRoundKey(round, state, RoundKey)
        }
        SubBytes(state)
        ShiftRows(state)
        AddRoundKey(Nr, state, RoundKey)
    }
    Orthogonality(state)
    removeInterleafBlocks(state)
}

#CTR
KeyExpansion(key);
AES_ECB_encrypt(key, plaintext, IV) {
    for(block in 0 to length/blocklength) {

        //encrypts IV
        interleafBlocks(IV)
        Orthogonality(state)

        Cipher(state, RoundKey) {
            AddRoundKey(0, state, RoundKey);
            for(round in 1 to Nr) {
                Bitslice_SubBytes(state)
                        manually calculate value for each byte
                        y14 = x3^x5
                        y13=x0^x6
                        y9=x0^x3
```

```
                        ...
                        state[0]=s7
                ShiftRows(state)
                MixColumns(state)
                AddRoundKey(round, state, RoundKey)
            }
            SubBytes(state)
            ShiftRows(state)
            AddRoundKey(Nr, state, RoundKey)
        }
        Orthogonality(state)
        removeInterleafBlocks(state)

        XorWithPT(output, plaintext);
        iv += 1;

        //move to next block of data to encrypt
        plaintext+=blocklength;
    }
}
```

8.3.4 T-Tables

```

P[0-3] contains plaintext
w = key

#ECB
TableGen(T0,T1,T2,T3)
inverted_TableGen()
KeyExpansion(key, KEY_LENGTH>>5)


AES_Encryption()

        xor input with roundkey
                        in0 = P[0] ^ w[0];
                        in1 = P[1] ^ w[1];
                        in2 = P[2] ^ w[2];
                        in3 = P[3] ^ w[3];

        use T-tables to calculate round outcome
                        //1Round
                        out0 = Te0[in0 >> 24] ^ Te1[(in1 >> 16) & 0xff] ^ Te2[(in2 >> 8) &
                        ↪  0xff] ^ Te3[in3 & 0xff] ^ w[4];
                        out1 = Te0[in1 >> 24] ^ Te1[(in2 >> 16) & 0xff] ^ Te2[(in3 >> 8) &
                        ↪  0xff] ^ Te3[in0 & 0xff] ^ w[5];
                        out2 = Te0[in2 >> 24] ^ Te1[(in3 >> 16) & 0xff] ^ Te2[(in0 >> 8) &
                        ↪  0xff] ^ Te3[in1 & 0xff] ^ w[6];
                        out3 = Te0[in3 >> 24] ^ Te1[(in0 >> 16) & 0xff] ^ Te2[(in1 >> 8) &
                        ↪  0xff] ^ Te3[in2 & 0xff] ^ w[7];

    ... hardcoded rounds

```

```
28          sbox lookup and shift for final round
29                          P[0] = (s[out0 >> 24] << 24 | s[(out1 >> 16) & 0xff] << 16 | s[(out2
                            ↪  >> 8) & 0xff] << 8 | s[(out3 & 0xff)]) ^ w[40];
30                          P[1] = (s[out1 >> 24] << 24 | s[(out2 >> 16) & 0xff] << 16 | s[(out3
                            ↪  >> 8) & 0xff] << 8 | s[(out0 & 0xff)]) ^ w[41];
31                          P[2] = (s[out2 >> 24] << 24 | s[(out3 >> 16) & 0xff] << 16 | s[(out0
                            ↪  >> 8) & 0xff] << 8 | s[(out1 & 0xff)]) ^ w[42];
32                          P[3] = (s[out3 >> 24] << 24 | s[(out0 >> 16) & 0xff] << 16 | s[(out1
                            ↪  >> 8) & 0xff] << 8 | s[(out2 & 0xff)]) ^ w[43];
```

9 Appendix B

9.1 Power measurements and TVLA graphs

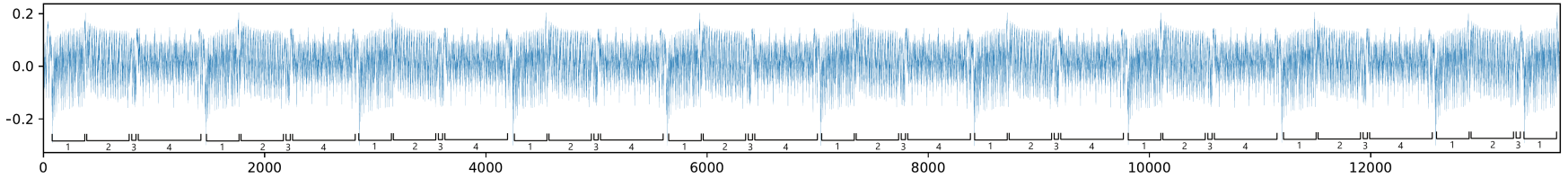


Figure 52: Tiny-AES - ECB - single trace

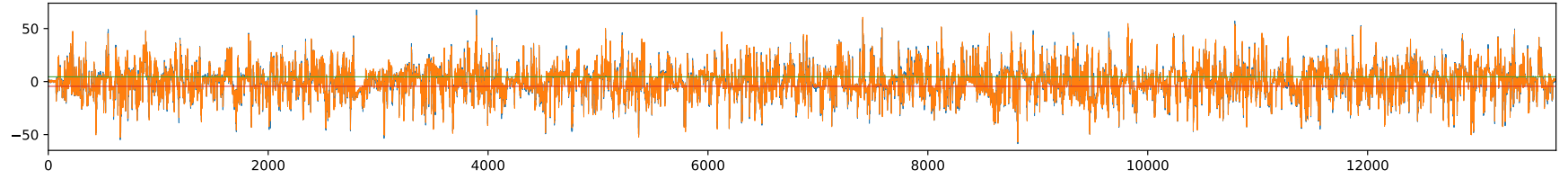


Figure 53: Tiny-AES - ECB - TVLA

Annotations:

1=Add round key 2=Substitution box 3=Shift rows 4=Mix columns

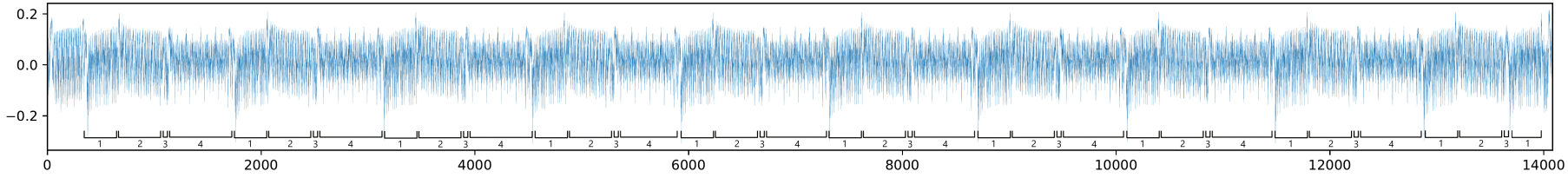


Figure 54: Tiny-AES - CBC - single trace

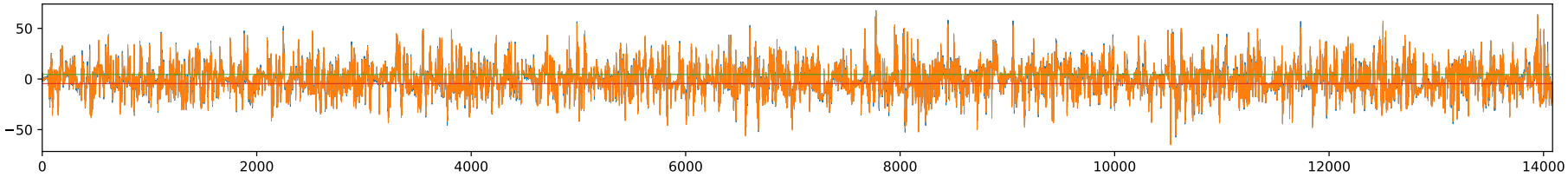


Figure 55: Tiny-AES - CBC - TVLA

Annotations:

1=Add round key 2=Substitution box 3=Shift rows 4=Mix columns

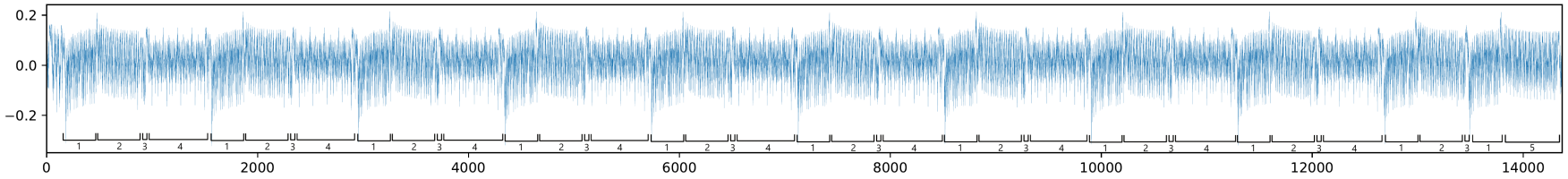


Figure 56: Tiny-AES - CTR - single trace

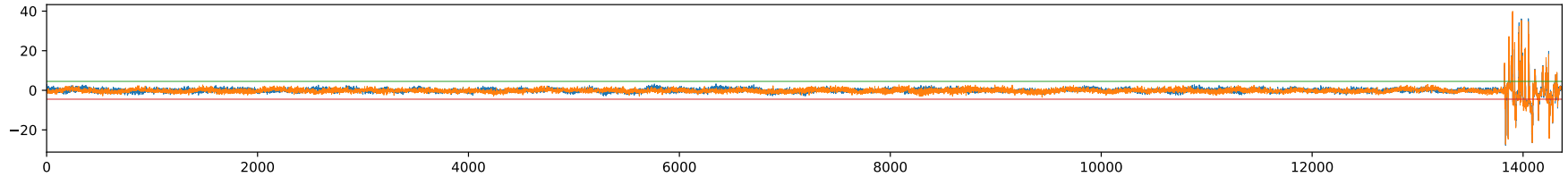


Figure 57: Tiny-AES - CTR - TVLA

Annotations:

1=Add round key 2=Substitution box 3=Shift rows 4=Mix columns 5=Increment IV & XOR plaintext

Masked-AES - ECB

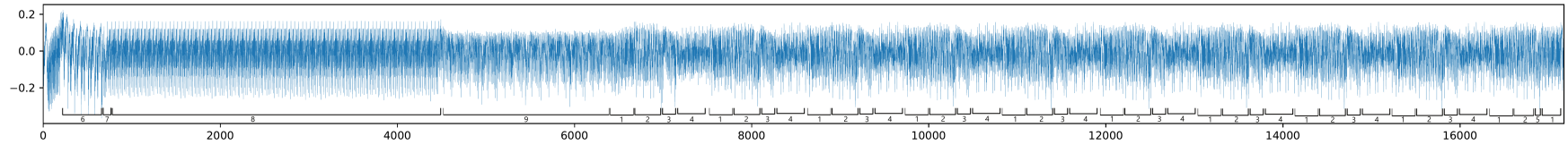


Figure 58: Masked-AES - ECB - single trace

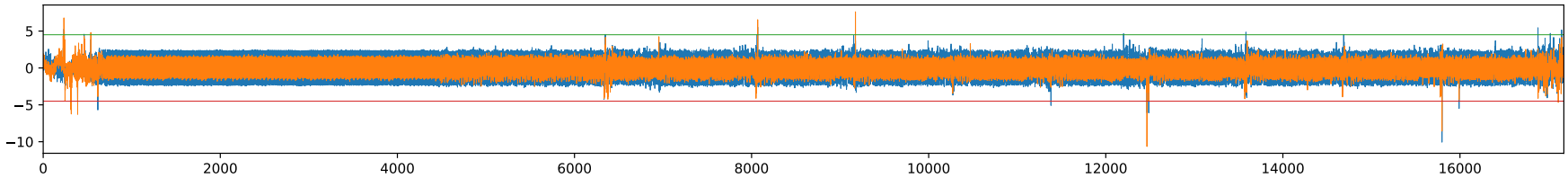


Figure 59: Masked-AES - ECB - TVLA

Annotations:

1=Add round key 2=Substitution box 3=Shift rows and remask 4=Mix columns 5=Shift

6=Randomly generate masks 7=Calculate MixColumn masks 8=Calculate Sbox masks 9=Mask change from M1' M2' M3' M4' to M

Masked-AES - CBC

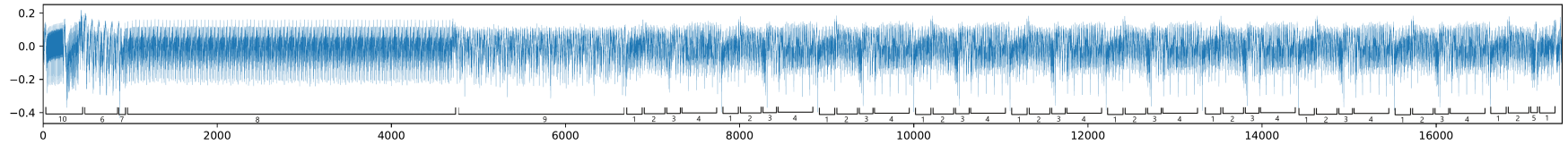


Figure 60: Masked-AES - CBC - TVLA

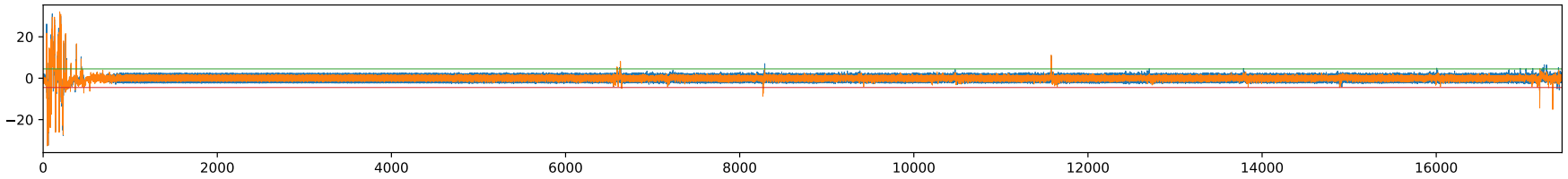


Figure 61: Masked-AES - CBC - TVLA

Annotations:

1=Add round key 2=Substitution box 3=Shift rows and remask 4=Mix columns 5=Shift

6=Randomly generate masks 7=Calculate MixColumn masks 8=Calculate Sbox masks 9=Mask change from M1' M2' M3' M4' to M

10=XOR with IV & Memory copies

Masked-AES - CTR

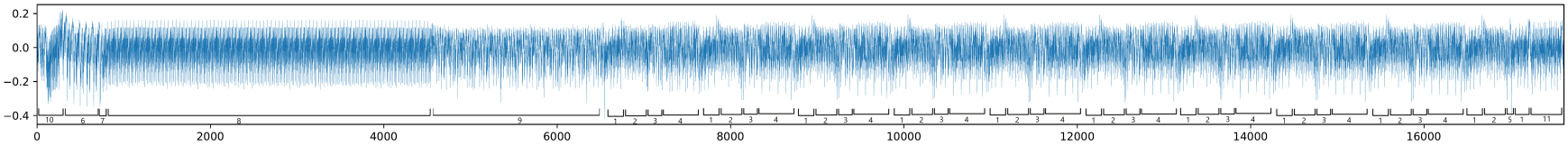


Figure 62: Masked-AES - CTR - single trace

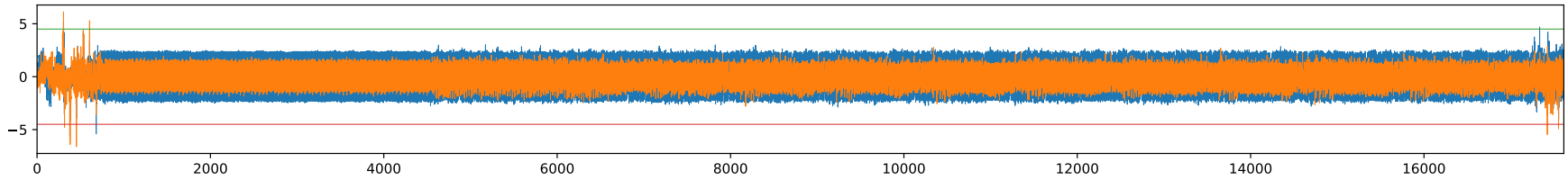


Figure 63: Masked-AES - CTR - TVLA

Annotations:

1=Add round key 2=Substitution box 3=Shift rows and remask 4=Mix columns 5=Shift

6=Randomly generate masks 7=Calculate MixColumn masks 8=Calculate Sbox masks 9=Mask change from M1' M2' M3' M4' to M

10=XOR with IV & Memory copies 11=Increment IV & XOR with plaintext

PQClean - ECB

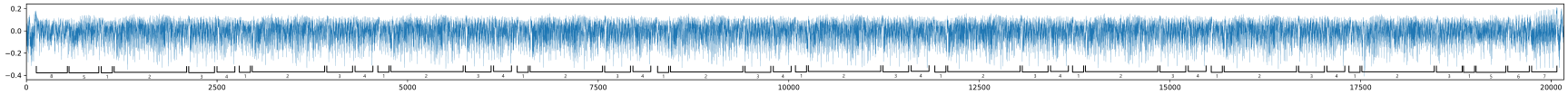


Figure 64: PQClean - ECB - single trace

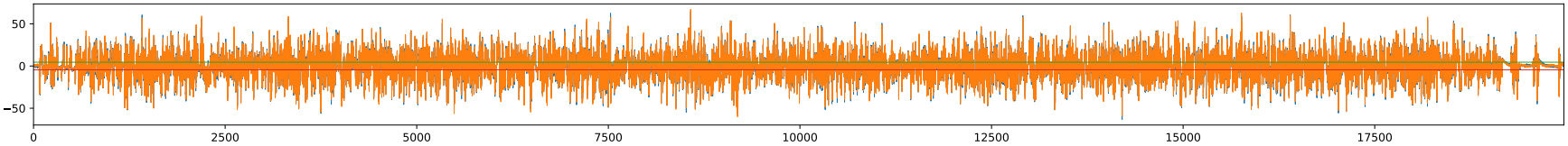


Figure 65: PQClean - ECB - TVLA

Annotations:

1=Add round key 2=Sliced substitution box 3=Shift rows 4=Mix columns 5=Create orthogonality 6=Interleave-out plaintext 7=Move to output 8=Interleave-in plaintext

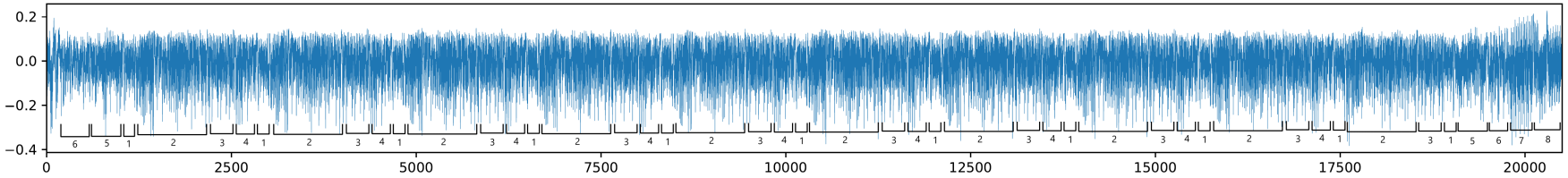


Figure 66: PQClean - CTR - single trace

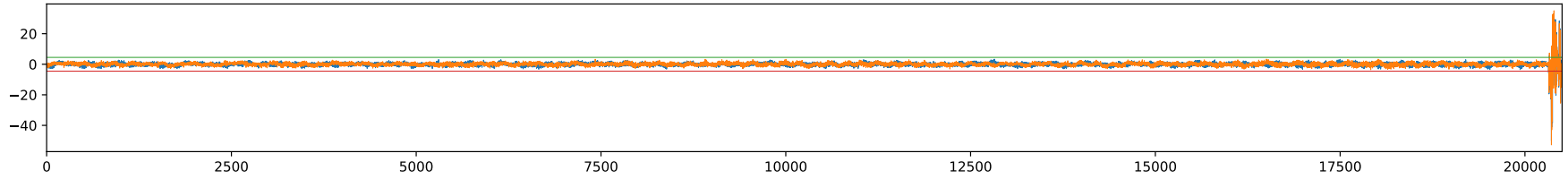


Figure 67: PQClean - CTR - TVLA

Annotations:

1=Add round key 2=Sliced substitution box 3=Shift rows 4=Mix columns 5=Create orthogonality 6=Interleave-out plaintext 7=Move to output 8=Interleave-in plaintext

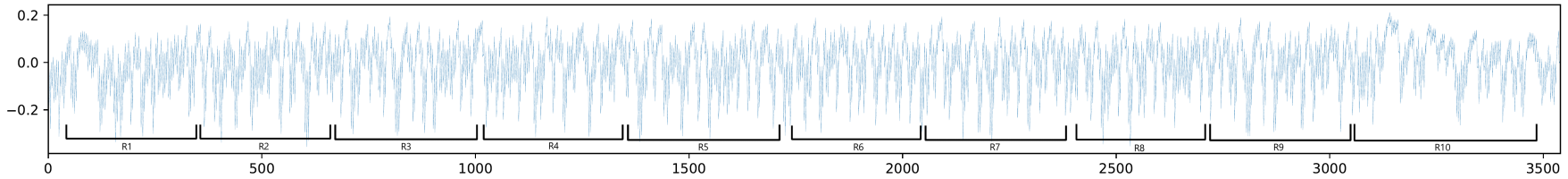


Figure 68: T-Tables - ECB - single trace

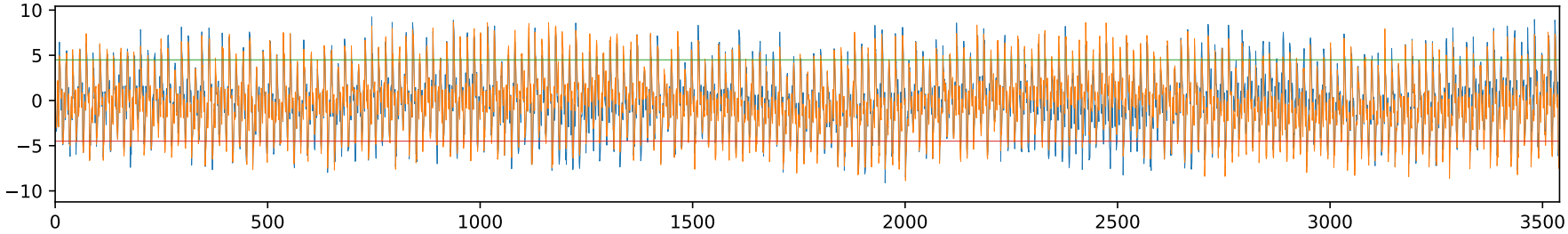


Figure 69: T-Tables - ECB - TVLA

Annotations:

R1-R10 denote the round number. Due to the way T-tables are implemented it is not possible to separate the measurements in the same manner as the other implementations.

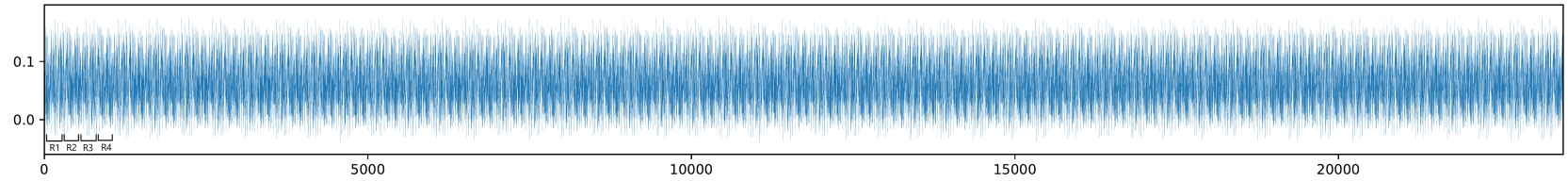


Figure 70: Tiny-AES - ECB - single trace - 1536 bytes

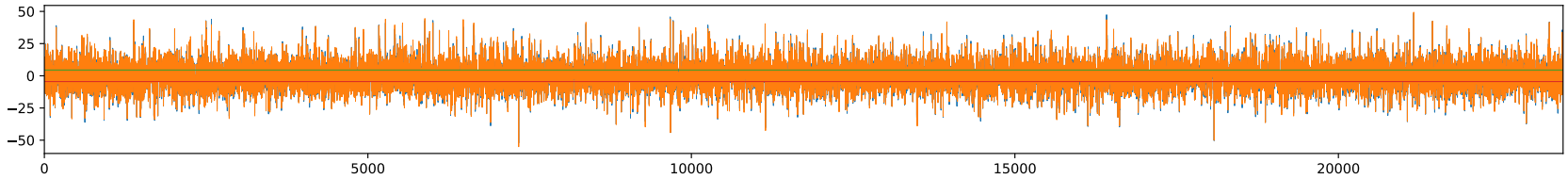


Figure 71: Tiny-AES - ECB - TVLA - 1536 bytes

Annotations:

R1-R4 are the new plaintext blocks. This scales all the way through to R96.

Large plaintext: Tiny-AES - CBC

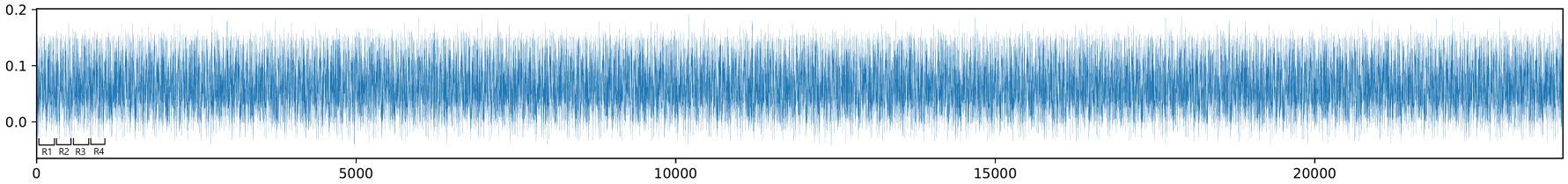


Figure 72: Tiny-AES - CBC - single trace - 1536 bytes

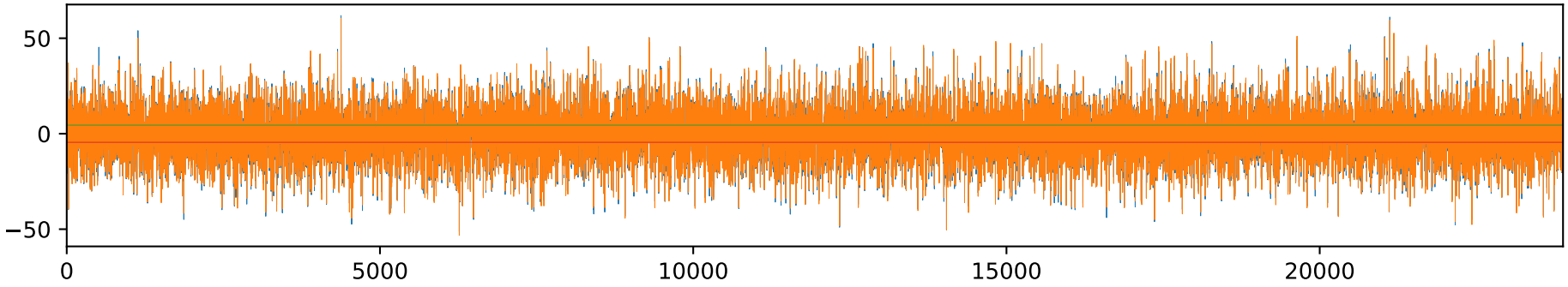


Figure 73: Tiny-AES - CBC - TVLA - 1536 bytes

Annotations:

R1-R4 are the new plaintext blocks. This scales all the way through to R96.

Large plaintext: Tiny-AES - CTR

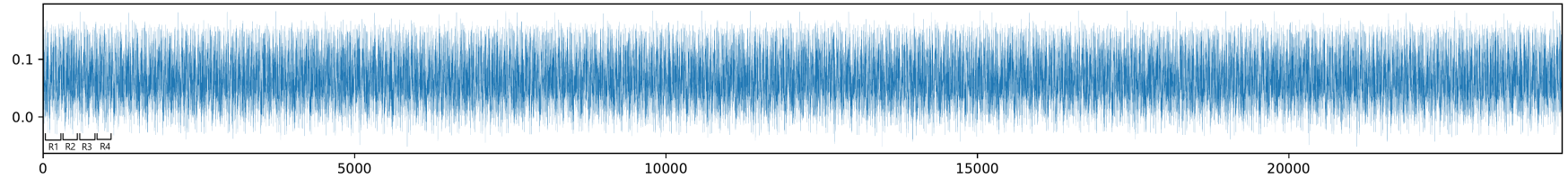


Figure 74: Tiny-AES - CTR - single trace - 1536 bytes

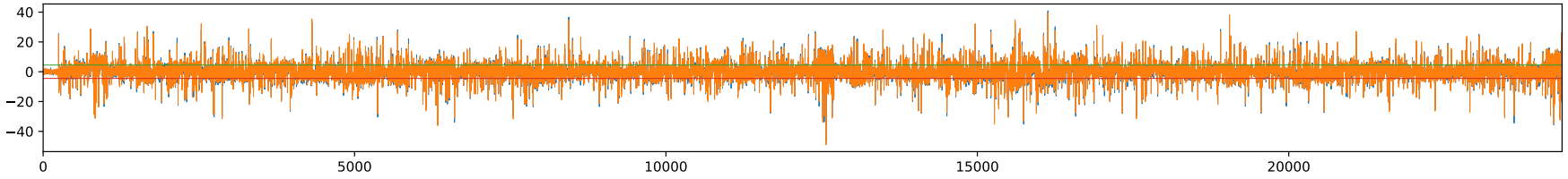


Figure 75: Tiny-AES - CTR - TVLA - 1536 bytes

Annotations:

R1-R4 are the new plaintext blocks. This scales all the way through to R96.

Large plaintext: Masked-AES - ECB

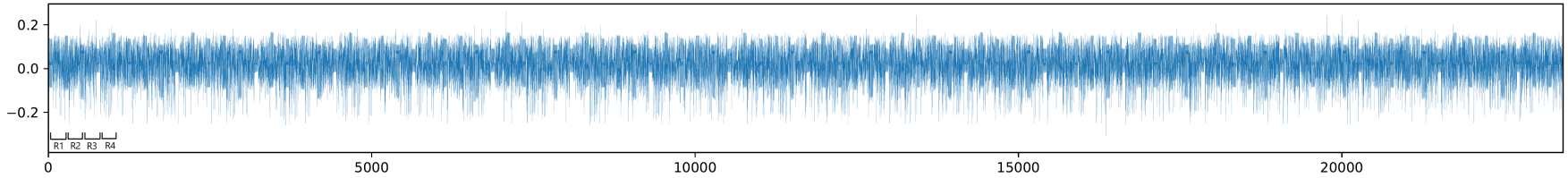


Figure 76: Masked-AES - ECB - single trace - 1536 bytes

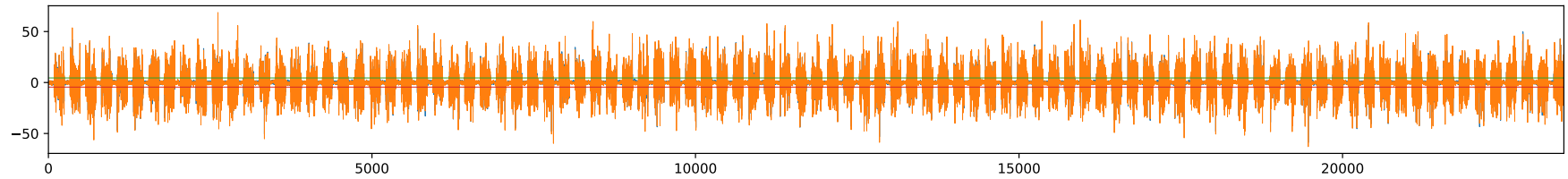


Figure 77: Masked-AES - ECB - TVLA - 1536 bytes

Annotations:

R1-R4 are the new plaintext blocks. This scales all the way through to R96.

Large plaintext: Masked-AES - CBC

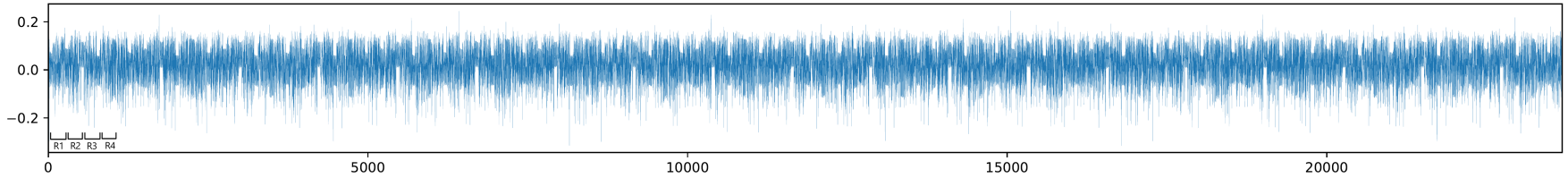


Figure 78: Masked-AES - CBC - single trace - 1536 bytes

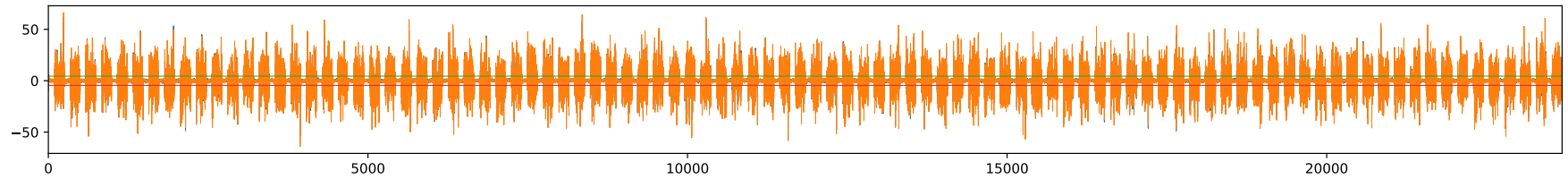


Figure 79: Masked-AES - CBC - TVLA - 1536 bytes

Annotations:

R1-R4 are the new plaintext blocks. This scales all the way through to R96.

Large plaintext: Masked-AES - CTR

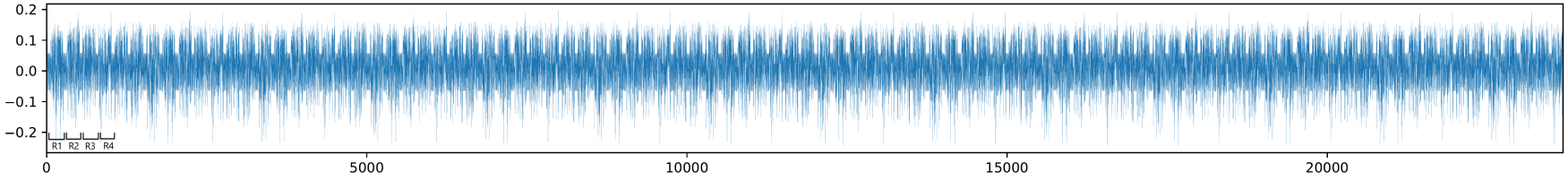


Figure 80: Masked-AES - CTR - single trace - 1536 bytes

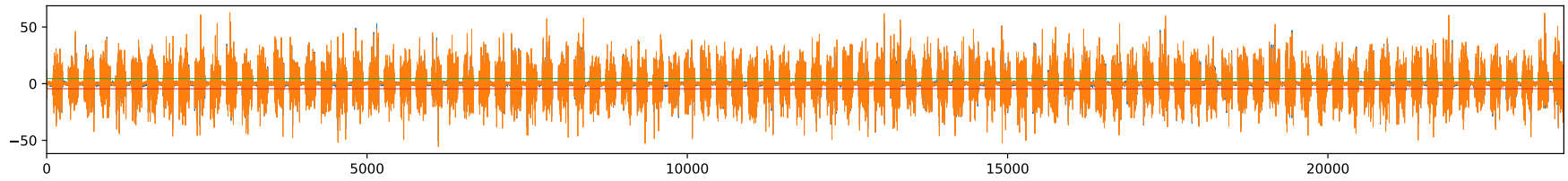


Figure 81: Masked-AES - CTR - TVLA - 1536 bytes

Annotations:

R1-R4 are the new plaintext blocks. This scales all the way through to R96.

Large plaintext: PQClean - ECB

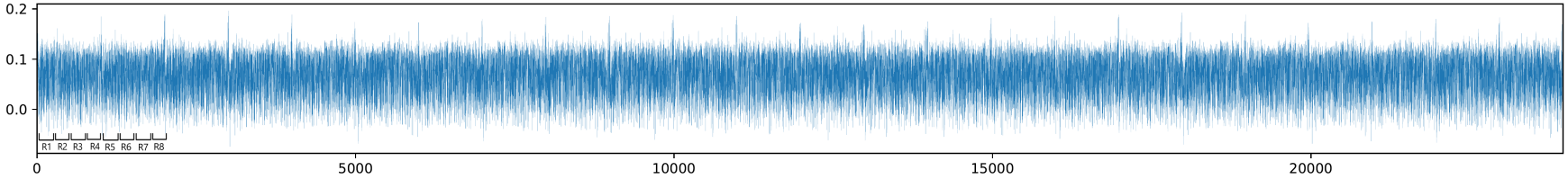


Figure 82: PQClean - ECB - single trace - 1536 bytes

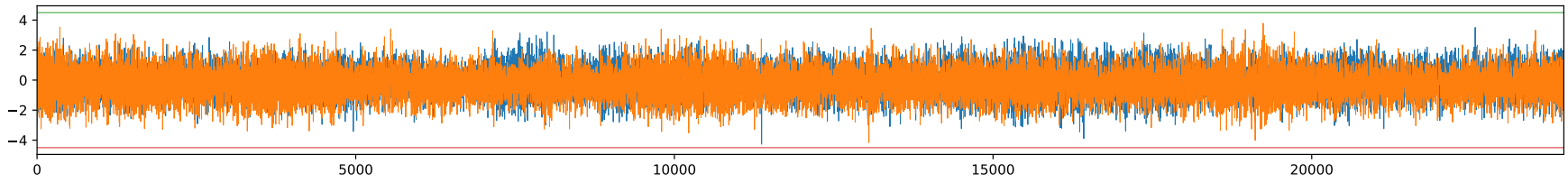


Figure 83: PQClean - ECB - TVLA - 1536 bytes

Annotations:

R1-R8 are the new plaintext blocks. This scales all the way through to R96.

Large plaintext: PQClean - CTR

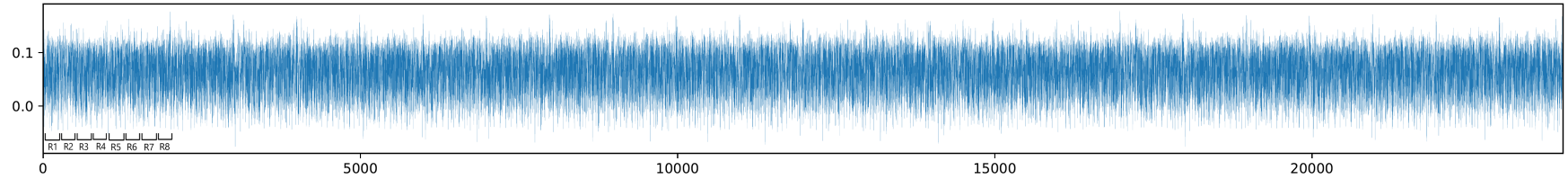


Figure 84: PQClean - CTR - single trace - 1536 bytes

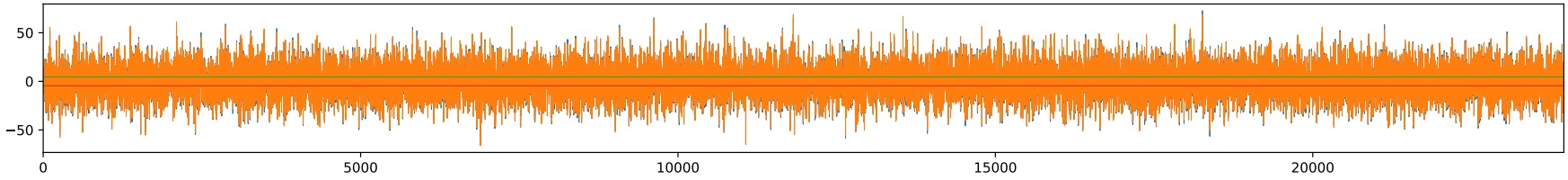


Figure 85: PQClean - CTR - TVLA - 1536 bytes

Annotations:

R1-R8 are the new plaintext blocks. This scales all the way through to R96.

Large plaintext: T-Tables - ECB

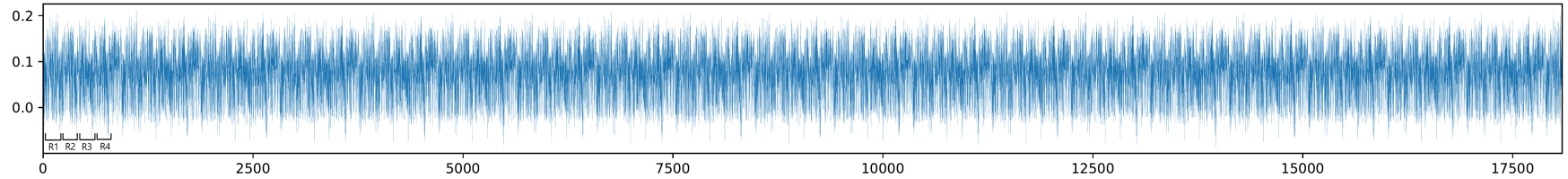


Figure 86: T-Tables - ECB - single trace - 1536 bytes

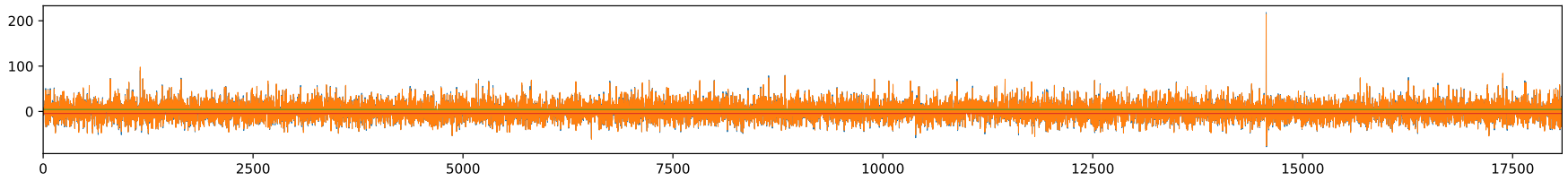


Figure 87: T-Tables - ECB - TVLA - 1536 bytes

Annotations:

R1-R4 are the new plaintext blocks. This scales all the way through to R96.