Evaluating the effectiveness of CPA and TA attacks on AES using information leakage

Fiske Schijlen (4947681), Alex Korpas (4907167), Stefanos Koffas (4915747), Chris Berg (4216776)

{f.h.schijlen, a.korpas-kamarianos, s.koffas, c.c.berg}@student.tudelft.nl

1 Introduction

AES, the world-wide used encryption scheme is assumed to be unbreakable by anything other faster than a brute-force attack. Therefore, the point of least resistance is not the encryption scheme itself, but the implementations of the scheme. In the first place the implementation could be wrong, resulting in reduced cryptographical strength. But even when AES is implemented correctly, the implementation of the scheme can leak information in the form of side channels. These could be either computational side channels like cache side channels or physical side channels, leaking information about the encryption to the real world. Examples of physical side channels include power consumption, heat production and electromagnetic attacks. In our work, we will focus on the latter type of side channels, specifically the power consumption side channel. This is the most powerful physical side channel, as side channels like heat production and electromagnetic radiation are either produced or related to the power consumption of the device. Monitoring the power consumption is in general the most invasive form of side channel attack of the three, requiring us to have direct contact with the processing unit. On the other side, power side channel is arguably the cheapest one to perform. This causes it to fit neatly in our scope as we will be performing our activities on a very limited budget.

In this report, we will show two different types of attacks using the power consumption side channel. We will compare performance of the attacks on both the traces we acquired and traces from other sources. We will implement two versions of AES: a vanilla, fast implementation and an implementation with counter measures. In section 2, we will be give a short overview of AES and the reasoning behind both our implementations of AES. In section 3, we will elaborate the attack models we will be using. In section 4, we will explain the setup of the trace retrieval part of our experiment. In section **??**, we will describe the metrics used for evaluating the performance of the attacks. These attacks will be elaborated in section 6, after which the performance of the attacks is revealed in section 7. We will wrap up with a discussion in section 8 and a conclusion in section 9.

2 AES implementation

2.1 Basic implementation

Advanced Encryption Standard is an relatively secure encryption scheme established by the U.S. National Institute of Standards and Technology (NIST) in 2001 [6]. In the present project, AES-128 was used, namely the block size and the key size that are used are both 128 bits, and it is consist of the following steps:

1. KeyExpansion: round keys are derived from the cipher key using Rijndael's key schedule. AES requires a separate 128-bit round key block for each round plus one more.
2. Initial round key addition:
   (a) AddRoundKey: each byte of the state is combined with a block of the round key using bitwise xor.
3. Repeat for 9 rounds:
   (a) SubBytes: a non-linear substitution step where each byte is replaced with another according to a lookup table.
   (b) ShiftRows: a transposition step where the last three rows of the state are shifted cyclically a certain number of steps.
   (c) MixColumns: a linear mixing operation which operates on the columns of the state, combining the four bytes in each column.
   (d) AddRoundKey
4. Final round:
   (a) SubBytes
   (b) ShiftRows
   (c) AddRoundKey

From all of the aforementioned steps, it is noteworthy to explain further the *SubBytes* procedure, since it is the one used from the side channel attacks described on the section 3. In the *SubBytes* step, each byte $a_{i,j}$ of the state block is transformed to $b_{i,l} = SBOX(a_{i,j})$ using a constant 8-bit substitution box $SBOX$. This step provides the non-linearity property to the algorithm.

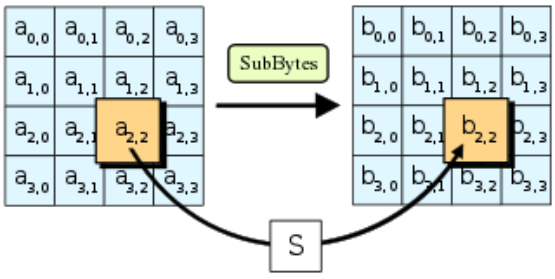


Fig. 1: SubBytes procedure of AES scheme

For the experimental part of the project, a version of AES-128 developed by Atmel[1] was used and modified in order to be installed on the Arduino UNO board that is used as target device.

2.2 Countermeasures

Both attacks described on section 3 work because of the similarity between previously executed encryptions and any newly executed encryptions. Therefore, the attacks can be thwarted by implementing countermeasures that make the encryptions' power traces indistinguishable from each other.

Countermeasure 1 - Dummy operations to hide SBOX traces As a first countermeasure we chose a simple hiding method that perform dummy accesses on the SBOX that are not used by the AES algorithm, but they generate random traces that aim to confuse the attacker. As explained above, the SBOX operation is the leakage source of the AES algorithm, performed on the SubBytes procedure. Thus, we implemented a modified version of that procedure that before calculation the actual operation $SBOX(state_k[i])$, it performs a random number of arbitrary similar operations. The pseudocode of the present countermeasure is presented on algorithm 1.

Algorithm 1 Hiding countermeasure

```
1: procedure SubBytes_cm(state_k[i])
2:     repeat
3:         randNumber ← random number from 0 to 1000
4:         SubBytes(randNumber  mod 128)
5:     until randNumber is odd
6:     SubBytes(state_k[i])
```

Countermeasure 2 - The transformed Masking Method This implementation is based in the transformed masking method of AES algorithm that is defined in [1]. In this approach, the message is masked at the beginning of the algorithm and it is unmasked at the end. Similarly to the original masking method, `XOR` operation is used; however, for the non-linear part of the algorithm the mask is arithmetic on GF(2^8) and not boolean.

The only non-linear part of the algorithm is the `sbox` which consists of two transformations; a multiplicative inversion in GF(2^8) and an affine transformation f which is applied on each byte $A_{i,j}$ of the input A (figure 2).

The modified `SubBytes` transformation is shown in figure 3. In this process an 8-bit boolean mask is used ($X_{i,j}$) to mask the input ($A_{i,j}$). In order to use the

[1] https://github.com/avrxml/asf/tree/master/sam0/applications/aes_software_library_demo

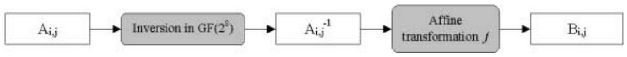


Fig. 2: The `SubBytes` transformation [1]

same boolean mask for the whole round it should be figured out how to generate the $A_{i,j}^{-1} \oplus X_{i,j}$. This procedure is depicted in detail in figure 1.

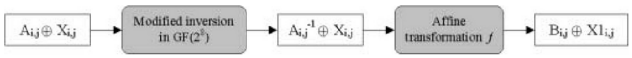


Fig. 3: The `SubBytes` transformation with masking countermeasure [1]

In this procedure the transformation of a boolean mask to a multiplicative one is shown. $Y_{i,j}$ is an 8-bit random number that is different than zero (multiplication with zero could break the implementation). The symbol $\bigotimes$ denotes the multiplication in GF(2^8) using the irreducible polynomial $m(x) = x^8+x^4+x^3+x+1$ as modulus.

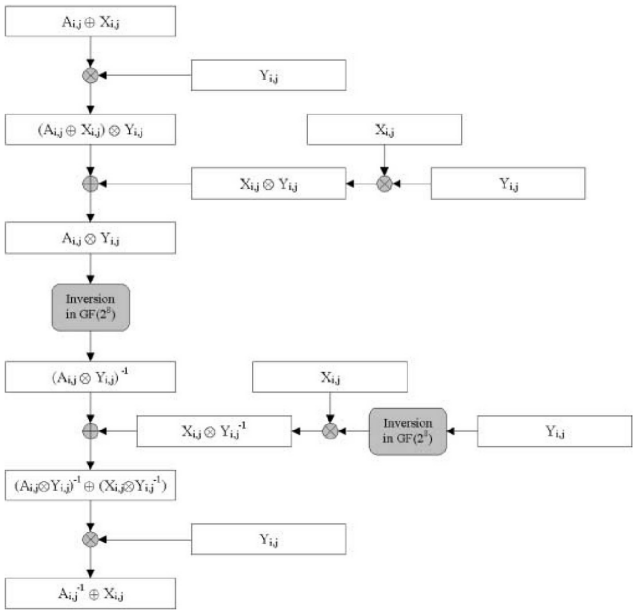


Fig. 4: Modified inversion in GF(2^8) with masking countermeasure [1]

Finally, in figure 5 the comparison of two different AES implementations is shown.

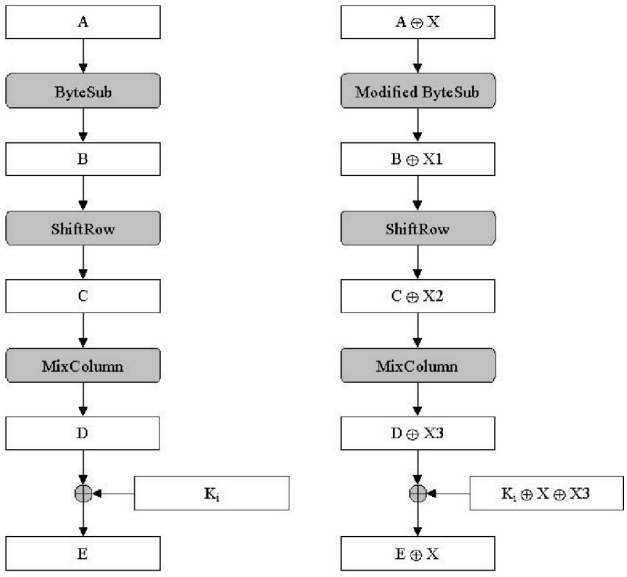


Fig. 5: Comparison of round i in AES without and with masking [1]

3 Attack descriptions

To obtain the full key used in the encryption performed by the target device, we execute a *correlation power analysis* (CPA) attack and a *template attack* (TA). These side-channel attacks have several similarities:

1. They make use of the information leakage found in the power consumption of the device running the AES algorithm.
2. They obtain the full private key by iteratively obtaining each of its 8-bit subkeys, where each subkey is obtained in the same way. They employ a brute-force method to obtain the subkeys to limit the attack's time complexity to $O(2^8 \cdot l)$, where l is the amount of bytes in the full private key.
3. When brute-forcing a subkey, they model the power that a device would consume when the AES algorithm executes the `subBytes()` phase of a round.
4. They are *chosen plaintext attacks*: to successfully obtain the key, they require power consumption traces that correspond to encryptions made for plaintexts that the attacker knows. This is because they use the encrypted plaintext to create a power consumption model, which can then be compared with the actual power consumption.

5. Each subkey is obtained by finding the most likely subkey candidate. The most likely candidate is determined by comparing the actual power consumption to the each possible modelled power consumption resulting from using a possible subkey.

We implemented both of the attacks in Python 3.

3.1 Correlation power analysis attack

We implemented the CPA attack proposed in [2] and based much of our implementation on the practical CPA approach detailed on the NewAE wiki.[9] This subsection will briefly explain how and why a CPA attack works.

A CPA attack is executed on power consumption traces that were generated by encrypting plaintexts known to the attacker. The attack uses these traces to obtain the i-th subkey byte of the key separately. For each possible subkey $k \in [0..255]$, it models the power that would be consumed when applying AES SBOX substitution right after XOR-ing the i-th plaintext byte with subkey guess k. It does this modelling for the i-th plaintext byte of all traces. It then compares these modelled power consumption values to the actual power consumption values by observing how they correlate with each other. The best subkey candidate is then the one whose modelled power consumption values correlate the most with the recorded values.

When computing the modelled consumption for a subkey guess k in the process of obtaining plaintext byte P_i, the attack computes $x = k \oplus P_i$ and applies SBOX substitution to obtain $y = sbox(x)$. By creating a power model for y and observing the correlation between the modelled values and the actual values, we are focusing on correlation between the two sets' respective points that correspond to the `subBytes()` part of each round. We model the power consumption for y by computing its *Hamming weight*, which is the amount of bits in y that is set to 1. By computing the modelled consumption as $HW(y)$, we are creating a model for the power it would take to go from a state where all bits are 0 to state y.

In several parts of the attack, we compute the correlation between values of two varying sets: (1) the set of power consumption values that were modelled using a subkey guess and some byte of all known plaintexts and (2) the set of actual power consumption values. We first compute the correlation between the modelled values and the actual values at a single point in all traces. By doing this for all points, we obtain the correlation value for the point that correlates most with the values modelled with subkey guess k. We compute this value for all $k \in [0..255]$ and take the maximum of those to determine the subkey of which the modelled consumption values correlate the most with the actual values.

3.2 Template attack

In a template attack, the adversary uses a large amount of traces with known keys to create *templates* either for each subkey guess or for groups of subkey

guesses. The adversary then obtains a smaller amount of traces of which he would like to know the used key. Finally, he obtains a subkey by determining to which subkey guess template the recorded traces are most likely to belong.

A notable difference between a template attack and a CPA attack is the requirement of the large amount of traces with which we can create templates. The key and plaintext with which these plaintexts were made has to be known to the adversary, so conventionally they would have to create the traces themselves. To do so, they need to have an exact copy of the target device, or else the templates will not make sense.

A template is defined as the combination of a set of means and a covariance matrix and can be used to create a multivariate normal distribution. Of course, creating such a template for each subkey guess using the entire set of points for all traces is unnecessarily expensive. To speed up the attack, we only construct a template for each possible byte Hamming weight $h \in [0..8]$ and only use a set of *points of interest* to create it. When the template traces are recorded, each of them consists of a large amount of points. The points of interest are the indices of a subset of those points that is found by selecting the points of which the power trace values differ the most, i.e. the points at which operations occur that significantly affect the power consumption. Traces are assigned to Hamming weight categories by creating a power model for the SBOX operation using a trace's corresponding plaintext and key.

In the actual attack, for each recorded attack trace, we create a power model for each subkey guess and use it categorise the trace. We can then check how likely it is that the trace matches the template and thus, how likely it is that this subkey guess was used. Doing this for all traces leaves the adversary with each subkey guess's probability of being used in the encryption of the targeted subkey byte.

The means and covariance matrix that comprise a template are computed by using the template category's traces' values at the designated points of interest. The means are just a collection of the mean power trace values corresponding to each point of interest. The covariance matrix describes the covariance between each pair of points' power trace values. The means and the covariance matrix can be used in the attack to create a multivariate normal distribution for each of the Hamming weight categories. For such a distribution, we can define a *probability density function* (PDF) that computes how likely it is that a given input value would be found in the computed normal distribution. After an attack trace is grouped into one of the Hamming weight categories for some subkey guess attempt, the PDF of the corresponding template's normal distribution is used to determine how likely the attack trace is to fit in with that template.

4 Acquisition of measurements

For our measurements, we used a mid-range RSDS 1102CML+ oscilloscope from RS Pro, available for around €300,-. This oscilloscope has a sampling rate of 1 Gigasamples per second. We measured the voltage over the ATMEGA328P-PU

microcontroller, with one probe at both VCC pins of the chip and one probe at the corresponding socket spaces for the VCC pins. We put a resistor between the chip socket and the VCC pins. We decided on a 10 Ohm resistor to get a certain amount of detail without getting too much of a power drop. We marked the first round of AES by setting the output of PB5-PD13 to 5 volts during the first round. This signal is send to the Trigger (or EXT) probe on the oscilloscope. Figure 6 shows a general picture of the setup of the experiment. Details about the exact connections of the wires can be found in the schematic in figure 7.

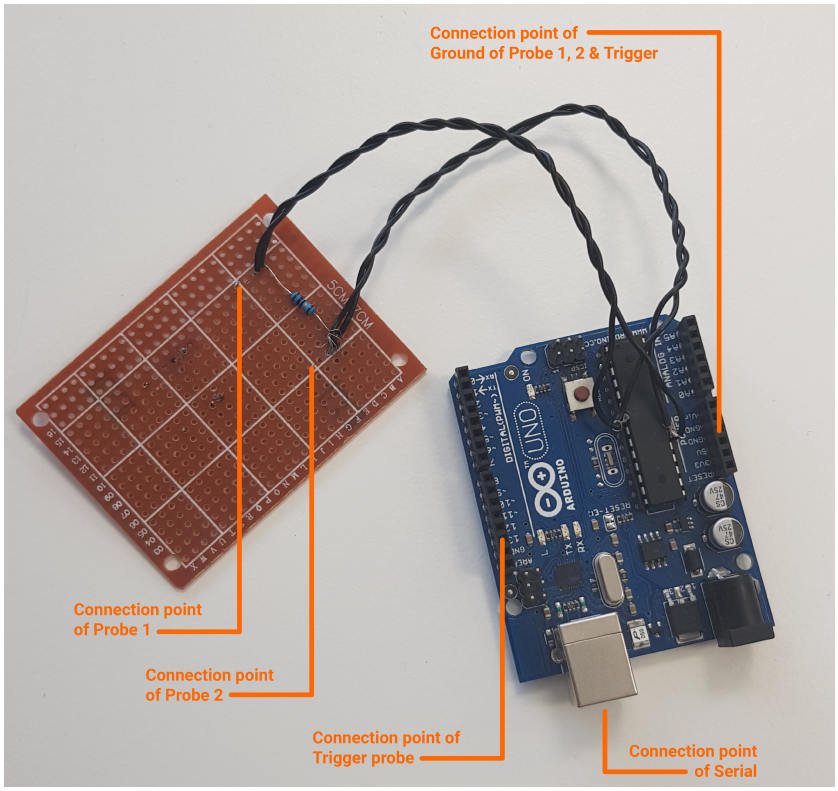


Fig. 6: The setup of the experiment

4.1 Trace retrieval

Initially, we used the Tektronix TBS1102B oscilloscope. We used Visa to communicate with the oscilloscope from within our python application, through a library called pyVisa. Unfortunately, we lost access to this oscilloscope before we could gather enough traces to perform CPA attack. Therefore, we continued

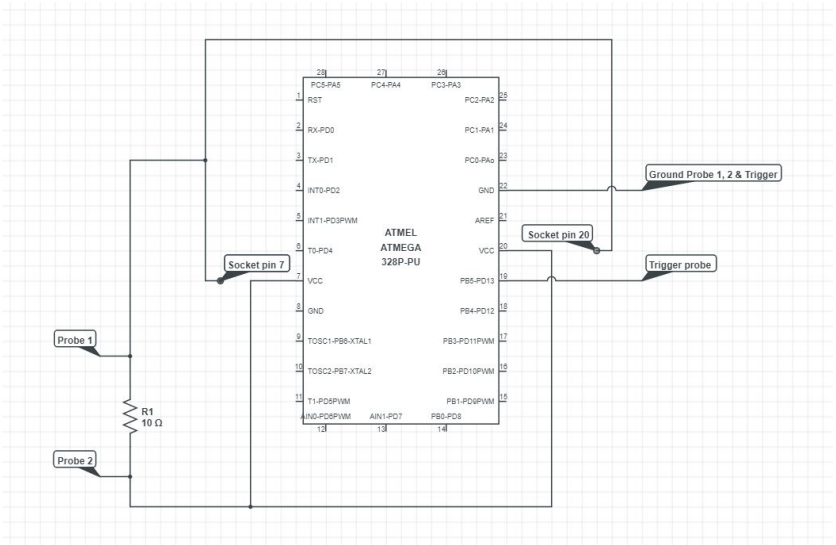


Fig. 7: Schematic of the microcontroller with experiment setup

on the RS Pro oscilloscope. This oscilloscope does not support the query commands from the versions of VISA we found. There was however a closed-source application available that was able to query the traces of this oscilloscope called EasyScope. We used a mouse macro to save traces from EasyScope, which we processed using our Python application running in the background.

The final trace acquisition loop consists of the following steps:

1. The python application waits for a file to be created in the traces directory.
2. On creation of a dummy file (by EasyScope) the application does two things:
   a. It sends a random plaintext of 16 characters to the Arduino, starting the encryption mechanism
   b. After a small delay, it saves the content of the screen of the oscilloscope in EasyScope, using a mouse macro.
3. After the file has been saved by EasyScope, the python application renames the file containing the power trace to the corresponding random plaintext.

This loop takes about 5 seconds to retrieve a single trace. After the power traces are acquired, the power trace from probe 2 is subtracted from the power trace of probe 1 to compensate for noise and fluctuations in the power input. The resulting power trace is then fed into the attack models, which will be elaborated in section 6.

For future use, the communication between EasyScope and the oscilloscope could be intercepted, and used to communicate from python to the oscilloscope directly. In that case, the communication to the oscilloscope could be done from within the python application itself, rendering the mouse macro obsolete.

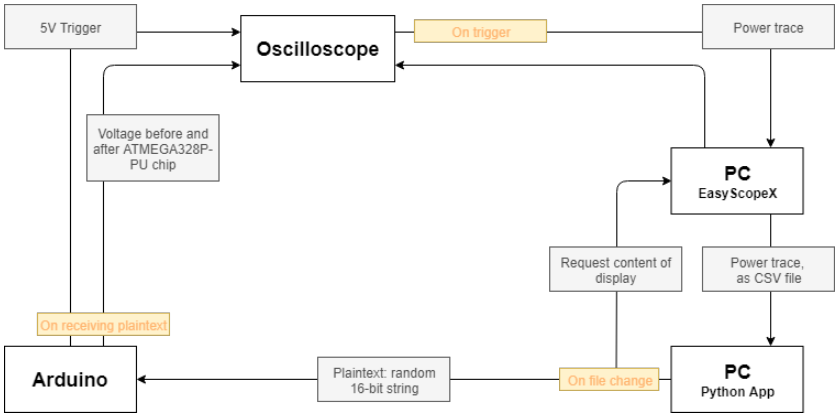


Fig. 8: Trace acquisition loop using EasyScope

4.2 Traces from an existing data set

We were unable to capture a sufficiently large amount of traces to successfully execute a template attack. To be able to perform experiments involving a template attack anyway, we obtained the freely available traces from TeSCASE[3] (Testbed for Side Channel Analysis and Security Evaluation). Specifically, we obtained 21000 traces originating from a standard AES implementation and 21000 traces originating from an implementation with *masking* applied as a countermeasure. Each trace consists of 3125 trace points.

These traces have previously been used to successfully evaluate side-channel power analysis methods in [7] and [4], which implies that they are apt to use in our experiments.

5 Guessing Entropy

The guessing entropy [5] is the most popular metric that is used for the evaluation of an attack. Guessing entropy expresses the number of guesses that an attacker has to make in order to find the secret piece of information of an algorithm. Thus, the minimisation of the guessing entropy is the aim of every attack. If the secret information can be broken to pieces, like the key that is used in AES (16 sub-keys for AES-128), then the partial guessing entropy can be used. Partial guessing entropy expresses the number of guesses that an attacker needs to make in order to guess correctly every part of the secret information (every subkey in case of AES).

6 Experiments setup

For both the correlation power analysis attack and the template attack, we want to observe the following:

1. The influence of the amount of used traces on the attack's effectiveness;
2. The influence of the amount of trace points on the attack's effectiveness;
3. The effect of the countermeasures embedded in AES. For CPA, "countermeasures" refers to the usage of dummy operations. For TA, it refers to the implementation of the transformed masking method.

We will use guessing entropy as a measure to gauge the effectiveness of the attacks.

6.1 Influence of the amount of traces

We will measure the influence of the amount of traces on the guessing entropy for both the CPA attack and the template attack. These measurements will be taken using power traces that were produced by our AES implementation without countermeasures. The resulting data will help us compare the effectiveness of both attacks for different amounts of traces and will later aid us in comparing our default implementation versus our implementation with countermeasures.

Correlation power analysis attack The accuracy of a CPA attack on some subkey depends on the correlation between the values of the recorded traces at the point we're attacking and the modelled power consumption values. Those modelled values correspond to the the SBOX simulations for that subkey XORed with the corresponding plaintext bytes. With just one trace, there is no increase or decrease in the recorded values, so the attack can only guess the subkey. With 10 or fewer additional traces, the correct subkey guess might already produce a high correlation value, but some incorrect subkey guesses might still have high correlation values as well since there are only 9 possible Hamming weights for a byte. Still, the probability that the same subkey's power models keep correlating with the correct subkey's power models is fairly low. Due to this swift elimination process of incorrect subkey candidates, we expect the amount of attack traces to exponentially affect the guessing entropy in a positive way.

Template attack A template attack ultimately determines how closely a trace matches a template by using the probability density function corresponding to that template. This implies the amount of traces used to create accurate templates plays a large role in the success of the attack. Therefore we expect exponential correlation between the amount of used template traces and the guessing entropy.

As long as the set of template traces is comprehensive enough to cover all possible subkeys, we believe we will require only about 10 attack traces for the guessing entropy to converge to 0. Other than hitting this low milestone, we expect the attack traces' influence on the guessing entropy to be minimal.

6.2 Influence of the amount of trace points

In these experiments, we will vary the *step size*, which indicates how aggressively we subsampled each trace's points. For example, step size 2 means that we used every second trace point from each trace's array of points.

Correlation power analysis attack When finding the subkey guess for which the SBOX simulation correlates the most, a CPA attack first finds the correlation coefficient of the trace point that correlates the most. Taking fewer trace points means the `subBytes()` step will be less distinguishable, so we expect the guessing entropy to decrease as the amount of trace points increases.

Template attack Increasing the amount of points in the template traces should increase the accuracy of the the selected points of interest and the values that comprise each template. We expect the guessing entropy to decrease linearly as the amount of trace points increases.

6.3 Effectiveness of the implemented countermeasures

Countermeasure 1 - Dummy operations to hide SBOX traces Hiding can take place in two domains; in time and in amplitude. Hiding in the time domain tries to randomise the time of occurrence of a critical operation and hiding in amplitude tries to reduce the effect of a specific operation to the power trace [8].

In our case we decided to randomise the time of occurrence of the `sbox` accesses to reduce the correlation between specific points in time. Thus, a random number of dummy instructions were run before `sbox` table is accessed. With this countermeasure and given that the number of power traces that are used remains unchanged, the guessing entropy of the attacker will increase due to the uncertainty that is introduced by the randomisation of the operations.

Countermeasure 2 - The transformed Masking Method In general, simple AES implementations contain non-linear operations (`sbox`) that can leak information about the key that is used. The attacker can take advantage of the fact that the dynamic energy consumption of an electronic device is proportional to the number of bits that are flipped from one state to another (from 0 to 1). Furthermore, in the first round of the AES the state is 0 so the bits that are flipped will be simply the Hamming weight of the new state after the `sbox` operation. For this reason it is convenient for an attacker to attack the first round, especially when the plaintext is known. However, if a random 8-bit mask is `XOR`-ed with the sensitive part of the algorithm, the leaked information from the power traces cannot lead to conclusions about the key that is used. Even if the attacker knows the plaintext, he will draw conclusions about the value $key \oplus mask$. If the `mask` is unknown no useful conclusions can be drawn. Thus, this countermeasure cannot be easily broken with first order attacks.

7 Results

7.1 Correlation power analysis attack

Influence of the amount of traces The impact of the amount of traces on the performance of CPA is displayed on figure 9. As it is clearly presented, the amount of traces and the guessing entropy present high negative correlation which implies that an increase on the number of the acquired traces causes a significant decrease on the guessing entropy, namely the performance is improved.

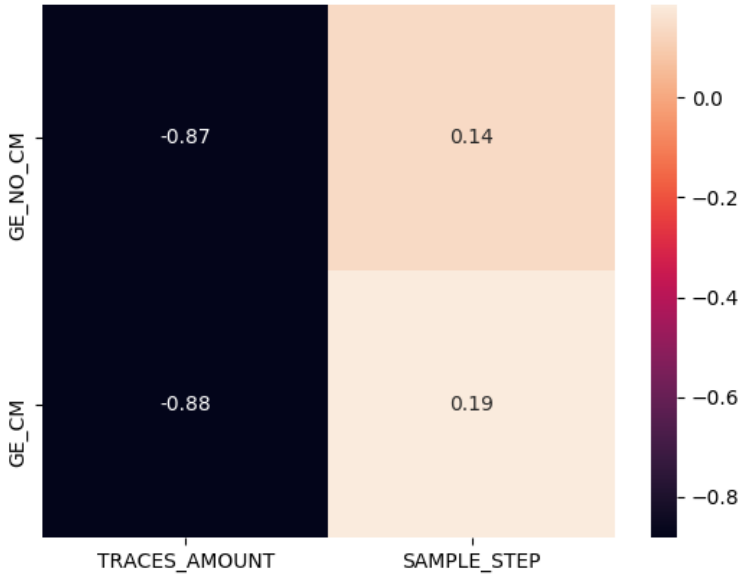


Fig. 9: Correlation of guessing entropy and amount of traces and trace points

Influence of the amount of trace points Regarding the effect of the amount of trace points, figure 9 present that there is a notable correlation with the performance of the CPA. Hence, as it is predicted above, a high number of trace points decreases the guessing entropy of the attack.

Effect of the countermeasures Figure 10 illustrates the different guessing entropy in between the implementations with and without countermeasures. A noteworthy observation is that the guessing entropy is relatively high even when

the amount of traces is over 700. That means that the key was not fully retrieved successfully on any of the attacks, probably because of poor quality of the acquired traces. Furthermore, we can see that there is not significant difference between the two implementations, as any difference is with the standard deviation range. Thus, we can conclude that the used hiding countermeasure is not effective.

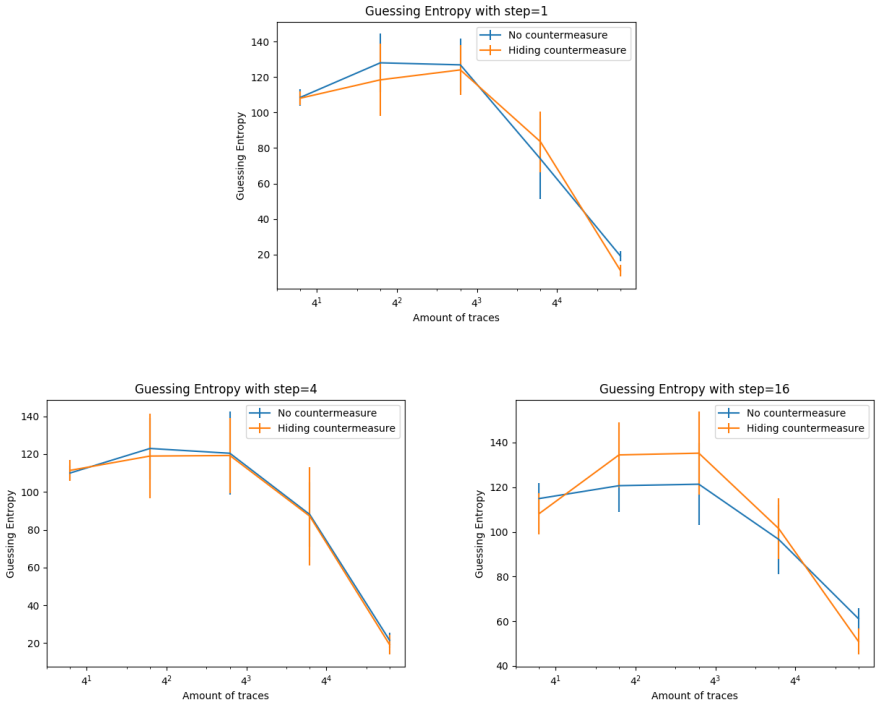
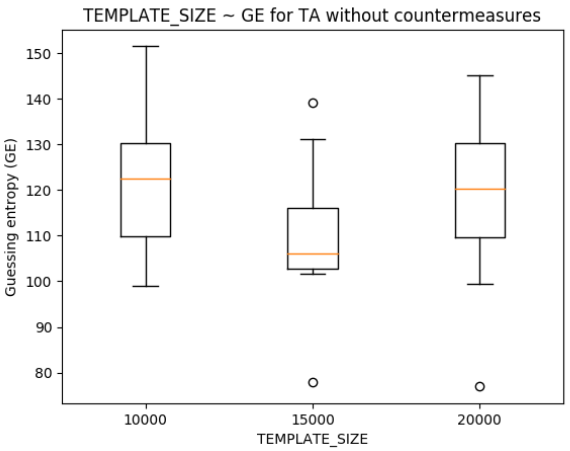


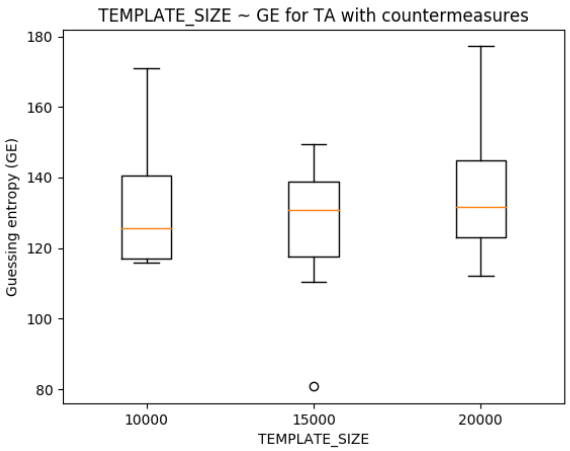
Fig. 10: Guessing Entropy of CPA for different sample step

7.2 Template attack

Influence of the amount of traces The effect of the amount of template and attack traces can be found in figures 11 and 12, respectively. To construct the plots from figure 11, we varied the amount of template traces, kept the step size constant at 1, and kept the amount of attack traces constant at 100. For the plots in figure 12, we kept the amount of template traces at 20000 and the step size at 1.

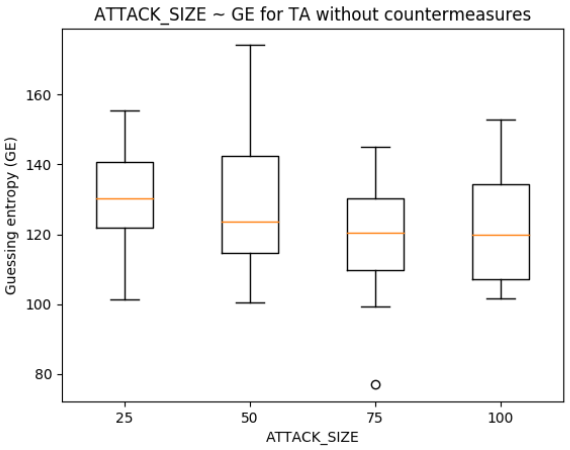


(a) Without countermeasures

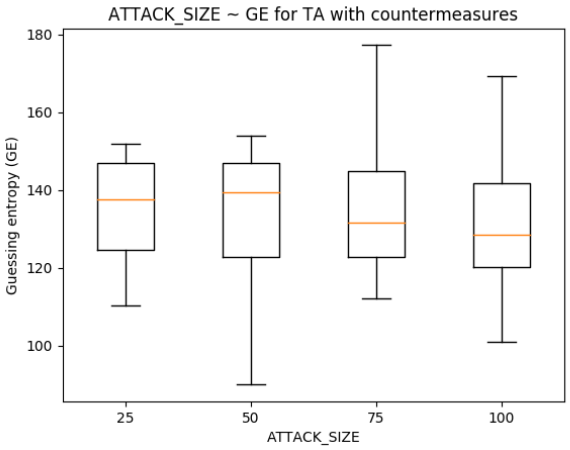


(b) With countermeasures

Fig. 11: Box plots indicating GE values for several amounts of *template* traces for a template attack.



(a) Without countermeasures



(b) With countermeasures

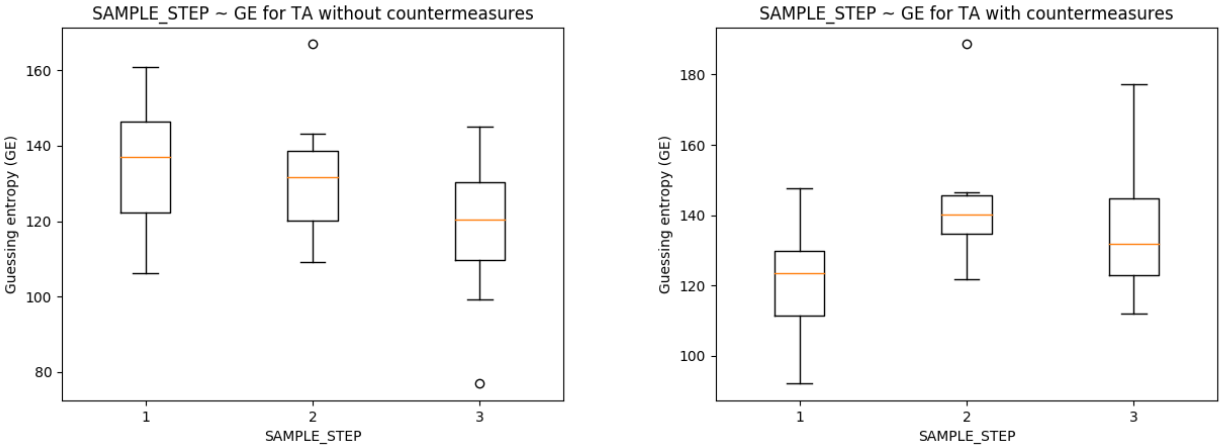
Fig. 12: Box plots indicating GE values for several amounts of *attack* traces for a template attack.

Both figures show that, regardless of countermeasures, increasing the amount of traces has no consistent effect on the GE. Furthermore, the boxes' whiskers indicate that the deviation of GE values is sufficiently large for us to infer that the amount of traces has no deterministic attack on a template attack's success on these traces.

However, this result is contrary to our hypothesis from section 6.1. Additionally, we observe high guessing entropy values for all amounts of traces, which is a counter-intuitive result for a template attack with these amounts. Due to this, we expect that either the traces or our methods have a flaw in them.

Influence of the amount of trace points Figure 13 displays the influence of the amount of trace points on the GE when applying a template attack to traces with or without countermeasures. For the traces we attacked with a TA, step sizes 1, 2, and 3 correspond to 3125, 1536, and 1042 trace points, respectively.

To construct these plots, we varied the step size, kept the amount of template traces constant at 20000, and kept the amount of attack traces constant at 100.



(a) Without countermeasures

(b) With countermeasures

Fig. 13: Box plots indicating GE values for several step sizes for a template attack. Step size

The figures display inversely related trends: the amount of trace points correlates positively with the GE for the unmasked traces, but correlates negatively with the GE for the masked traces. The positive correlation between the amount of trace points and the GE does not fit our hypothesis from section 6.2: more points should lead to the opportunity for better points of interest, so the GE should grow or remain static as the amount of trace points grows.

However, it should be noted that the boxes' deviation whiskers indicate that that the differing GE results within one trace set's results are negligible.

Effect of the countermeasures We expected the masking procedure to significantly increase the guessing entropy across the board. Most of the figures do not show such behaviour though. Figure 12 does show slightly higher median GE values for all amounts of attack traces, which means the countermeasures increase the GE for all cases with 20000 template traces and 3125 trace points, which are the maximum values for those variables.

However, due to the large GE value deviations in that figure's plots and the fact that similar patterns are not present in the other figures, we cannot conclude that the countermeasures consistently worsen the effectiveness of a template attack.

8 Discussion

8.1 Acquisition of measurements

During our experiments, we encountered a number of issues we had to overcome. When reproducing this experiment, it is important to take the following things in consideration:

- It is not recommended to use a generic breadboard, as this introduces too much noise. For our final setup we soldered the resistor to a proof board and made special connection points for the probes of the oscilloscope.
- In order for the attacks to proceed, it is of vital importance to link the plaintexts to the right power traces. We started of with the Arduino producing the plaintexts through a pseudo-random function using a known seed. This did however cause a lot of inconveniences in terms of synchronising the Arduino with our trace acquisition application. For our final setup, we sent the plaintexts through the serial input of the Arduino.
- Measure directly on chip. At first, we powered the Arduino through the VCC pin on the Arduino board and measured the voltage on that pin. We started with this approach, as this higher level of measuring is a less intrusive way to measure the power consumption and is therefore more likely to be easily applicable to a real world scenario. Unfortunately, this way of measuring did not provide us with useful traces. We ended up measuring the voltage on both VCC pins on the ATMEGA microcontroller. Not only did this approach provide us with useful traces, it also simplified the setup of the experiment as we were able to communicate directly to the chip through the serial port of the Arduino.
- We recommend using the Visa protocol for communicating with the oscilloscopes. Before the Tektronix TBS1102B oscilloscope became unavailable for us, we used pyVisa to gather traces directly within the python application. Without the need for an external program, we were able to take about twice as many traces per minute.

– The serial of the Arduino can be very easily flooded. On retrieval of a plaintext, we responded with 10 lines of debug information of approximately 20 ASCII characters long. This congested the serial after exactly 65 traces. After disabling all debug information, the Arduino ran for an indefinite period.

8.2 Countermeasures

Hiding This countermeasure seems to be relatively simple because an adversary can break it simply by using more power traces to overcome the uncertainty that was introduced by the random number of dummy operations that were introduced before accessing the `sbox` table.

Masking If the `mask` is unknown no useful conclusions can be drawn by an attacker. However, a higher order attack that uses both the `plaintext` and the `mask` can easily break this implementation.

8.3 Experiments

Our TA experiments show no consistent trends. This is likely due to a flaw in our methodology or the traces we used. To get an initial set of conclusive results, we would like to obtain a sufficiently large amount traces using the method we described in section 4.1 and re-run our TA experiments with them.

Furthermore, because our CPA attack is currently unable to find the complete key, we would like to attempt to reduce the noise in the traces we obtain from the oscilloscope. One way to do this might be through using another resistor in our setup instead of the one we use now.

9 Conclusion

With our CPA experiments, we determined that that amount of traces significantly contributes to the guessing entropy, regardless of whether the AES implementation inserts dummy operations or not. Figure 9 shows this in a compact manner.

The TA experiments' results show no consistent trends and contain very high standard deviations for the observed GE values. Because of these aspects, we cannot infer anything regarding the variables' influence on the GE for template attacks, or make any certain statements regarding the effectiveness of AES masking in defending against template attacks.

References

1. Akkar, M.L., Giraud, C.: An implementation of des and aes, secure against some attacks. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems — CHES 2001. pp. 309–318. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)

2. Brier, E., Clavier, C., Olivier, F.: Correlation power analysis with a leakage model. In: Joye, M., Quisquater, J.J. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2004. pp. 16–29. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
3. Fei, Y.: Power trace data (2019), http://tescase.coe.neu.edu/?current_page=POWER_TRACE_LINK
4. Luo, P., Fei, Y., Zhang, L., Ding, A.A.: Side-channel power analysis of different protection schemes against fault attacks on aes. In: 2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14). pp. 1–6. IEEE (2014)
5. Standaert, F.X., Malkin, T.G., Yung, M.: A unified framework for the analysis of side-channel key recovery attacks. In: Joux, A. (ed.) Advances in Cryptology - EUROCRYPT 2009. pp. 443–461. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
6. of Standards, N.I., Technology: Advanced encryption standard. NIST FIPS PUB 197 (2001). https://doi.org/https://doi.org/10.6028/NIST.FIPS.197
7. Swamy, T., Shah, N., Luo, P., Fei, Y., Kaeli, D.: Scalable and efficient implementation of correlation power analysis using graphics processing units (gpus). In: Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy. p. 10. ACM (2014)
8. Tillich, S., Herbst, C., Mangard, S.: Protecting aes software implementations on 32-bit processors against power analysis. In: Katz, J., Yung, M. (eds.) Applied Cryptography and Network Security. pp. 141–157. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
9. Various: Correlation power analysis (2019), https://wiki.newae.com/Correlation_Power_Analysis