Timing Attack on a Modified Dynamic S-box Implementation of the AES InvSubBytes Operation

Johannes Obermaier, Tobias Laas and Markus Roner

Institute for Security in Information Technology
Technische Universität München
Arcisstr. 21
80333 Munich, Germany
{johannes.obermaier, tobias.laas, markus.roner}@tum.de

Abstract: This paper demonstrates a novel timing attack on a software implementation of the AES decryption algorithm. The implementation was optimized to reduce its code and memory footprint by utilizing an inverse S-box operation which directly calculates the substitution values instead of fetching them from a pre-computed look-up table. This code-size optimized implementation was created as part of a laboratory for which a smart-card emulator was designed and physically tested. Later on, we noticed that the implementation shows a data-dependent execution time for which we developed a novel timing attack. It is based on a timing-model which was derived from an analysis of the implementation. The feasibility of the approach was first proved by a simulation. The subsequent application of the attack on the smart-card emulator in a real setup was successful. This paper describes the analysis done to conduct the attack and emphasizes the dangers of incautiously implemented cryptographic algorithms.

1 Introduction

The Advanced Encryption Standard (AES) is one of the most widely used standards for symmetric encryption employed in today's embedded applications. The widely used cipher implementation, AES-128, uses a key length and block size of 128 bits and is considered to be secure against all cryptanalytic attacks known to this date. The attack published by Bogdanov et al. [BKR11] reduces the computational complexity from 2^{128} to $2^{126.1}$, speeding up the process although still not making it computationally feasible. Despite its mathematical security, hardware and software implementations of AES may be vulnerable to a different class of attacks known as Side-channel Analysis (SCA) attacks. These attacks are based on exploiting the non-intended leakage of information, which occurs during physical operation of the cipher, to retrieve its cryptographic secret. The first publications, which represent a cornerstone for what has become an important field of research in both academia and industry, correspond to Kocher's papers on timing attacks published in 1996 [Koc96] and on power-analysis published in 1999 [KJJ99]. Since then a vast number of publications have dealt with novel SCA attacks and their corresponding countermeasures. A comprehensive study of power-analysis on AES implementations

is given in [MOP07] by Mangard et al.. Timing attacks are usually treated separately to those related to power consumption. In 2004, Bernstein published the first cache-timing attack to break a server using OpenSSL's AES encryption [Ber04]. In that paper, the high-precision time-stamp of the server was used for the attack. In 2005, Osvik et al. [OST05] demonstrated cache-timing attacks on a PC platform kept under control, requiring only a few hundred measurements. In that attack, the timing information was extracted from the CPU's memory cache. While there are approaches to secure speed or memory usage optimized AES implementations against timing attacks on modern 64-bit CPUs [KS09], these methods are not fully applicable to the simpler processors used in embedded applications.

Smart-cards are widely used for applications ranging from authentication and encryption to any kind of payment. Common to all those applications is the requirement for a secure and reliable operation, while preventing the extraction and replication of sensitive data from the card's embedded integrated circuit. In the case of the pay-tv smart-card (emulator) in the laboratory, a key encryption key (KEK) is stored on the smart-card and used for decryption of a session key, which is then used for the video stream decryption. The smart-card emulator uses a standard, low-cost, 8-bit ATmega microcontroller, which was utilized for testing the implementation of the AES-128 and of the timing attack. Since the embedded system on the smart-card operates widely isolated from the environment, not many indications are left to deduce relevant timing information. It is a convenient way to use power traces in order to gain this information. To exclude unwanted influence from power-analysis deviations on the timing attack however, the use of a trigger signal for timekeeping on a pin of the microcontroller is often helpful and sufficient for a proof of concept.

In Sect. 2, we begin with the explanation of an alternative implementation of the `InvSubBytes` function. We continue with its weakness and the attack's theory, together with its simulation, in Sect. 3. The practical implementation of the attack is demonstrated in Sect. 4. Finally, the results are evaluated and countermeasures are discussed in Sect. 5.

2 An alternative approach for `InvSubBytes`

In the course of an AES-128 decryption, the non-linear byte substitution `InvSubBytes` is applied to the internal state of the decryption ten times. `InvSubBytes` consists of two steps: applying an affine transformation and taking the multiplicative inverse in the Galois Field $\mathrm{GF}(2^8)$ [Nat01]. Calculating the multiplicative inverse is computationally expensive. Traditionally, this substitution and its inverse, `SubBytes`, have been implemented on microcontrollers using a substitution table (S-box), as described in the AES specification [Nat01], because that avoids calculating the multiplicative inverse at run time and thereby increases performance. As described in [DR98], "there is some ROM/performance trade-off" in every implementation, i.e. if more execution time is spent, the amount of ROM needed could be reduced. The Rijndael AES Proposal also gives an example for an optimized implementation: "For implementations where RAM is scarce, the Round Keys can be computed on-the-fly [...]" [DR98] . As an answer to those reflections, we decided that in our implementation, not only the round

keys would be computed on-the-fly, but also the entries of the S-boxes. That way more flash memory and RAM is saved because the S-boxes' values do not need to be stored. This is advantageous for devices which are very restricted in memory. The difference is clearly visible. The traditional implementation in C with static S-boxes requires $2.13\,\text{kB}$ of flash memory on our microcontroller, but with the on-the-fly calculated S-box and inverse S-box, the demand decreases to only $1.69\,\text{kB}$, which is 20% less. This reduction in size comes with a performance trade-off. The execution time per decryption is increased from $20\,\text{ms}$ to $200\,\text{ms}$.

There are several algorithms to find the multiplicative inverse, including the extended Euclidean algorithm and the simple brute-force search we used in our implementation, c.f. Appendix A. There is a special case when computing the multiplicative inverse in `InvSubBytes`, as the inverse of 0x00, which can occur as result of the affine transform, does not exist. In this case, 0x00 has to be returned, as specified in [Nat01]. The algorithms are usually not executed in this case, but instead 0x00 is returned immediately, which causes a shorter execution time. The question arises whether this data-dependent execution time has a negative effect on security and if it can be exploited by a timing attack.

3 Timing attack

3.1 Theory

In this subsection, the theory of a timing attack on the implementation of the AES-128 decryption is described. As already mentioned in the previous section, the brute-force search algorithm for the computation of the multiplicative inverse described in Appendix A has a data-dependent execution time, which will be exploited in this timing attack.

In the attack, random ciphertexts were decrypted on the microcontroller and the plaintext vector p and the execution time t were recorded for each decryption. We know that in its last round

$$p = k \oplus \texttt{InvSubBytes}(s) \ , \tag{1}$$

where k is the key and s is the intermediate state vector before the invocation of `InvSubBytes`. If both the output of `InvSubBytes` and p are known, the key can be computed by rearranging the equation to

$$k = p \oplus \texttt{InvSubBytes}(s) \ . \tag{2}$$

For byte s_i of the state, where $i \in \{0, \ldots, 15\}$, this means that

$$k_i = p_i \oplus \texttt{InvSubBytes}(s_i) \ , \tag{3}$$

as `InvSubBytes` and $\oplus$ are byte-wise operations. As p_i is recorded, the idea of the following attack is to derive information about the output of $\texttt{InvSubBytes}(s_i)$ given the execution time t, in order to find out k_i.

Table 1: Probabilities for the number of zeros (a) and votes for the attack.

Zeros (a)	Probability	vote for value for p_i	vote for other values
0	53.5%	$\ln(1/10000)$	$\ln(9999/(10000 \cdot 255)$
1	33.5%	$\ln(1/160)$	$\ln(159/(160 \cdot 255))$
2	10.5%	$\ln(2/160)$	$\ln(158/(160 \cdot 255))$
3	2.2%	$\ln(3/160)$	$\ln(157/(160 \cdot 255))$
4	0.3%	$\ln(4/160)$	$\ln(156/(160 \cdot 255))$
5	0.04%	$\ln(5/160)$	$\ln(155/(160 \cdot 255))$
6	$< 0.01\%$	$\ln(6/160)$	$\ln(154/(160 \cdot 255))$

For one full AES-128 decryption, `InvSubBytes` is called ten times, excluding the key expansion, which does not induce timing differences, because the key is assumed to be constant. Each of those calls applies the transformation to each of the 16 bytes of the internal state, i.e. there are 160 calls to `GF2_8InvMult`, which performs the inversions in $\mathrm{GF}(2^8)$. If the byte at the input of `GF2_8InvMult` is zero, significantly less time is needed to compute its multiplicative inverse. In this case, the output of `InvSubBytes`(s_i) is zero and $k_i = p_i$. The remaining operations of the implementation have significantly less variation in execution time. Therefore the attack focuses on the number of zeros.

First, the execution time is analyzed and the number of zeros occurring at the input of `GF2_8InvMult`, a, is estimated. More about that in Sect. 4.

For random data which is decrypted, the probability for a zeros is

$$P_A(a) = \binom{160}{a} (1/256)^a (255/256)^{160-a} \quad , \tag{4}$$

under the assumption that the zeros are uniformly distributed over all invocations of `GF2_8InvMult`. The evaluated probabilities can be seen in Table 1.

It is unknown in which round and at which byte the zeros occurred, but the probability of n zeros appearing in the last round of the AES decryption, i.e. in the last invocation of `InvSubBytes`, given that there are a zeros in the whole decryption, is

$$P_{N|A}(n \mid a) = \frac{\binom{16}{n}\binom{144}{a-n}}{\binom{160}{a}} \quad . \tag{5}$$

When the information acquired from the measurements is combined, the zeros arising in the last round provide so-called correct information about the key and the zeros arising in the other rounds provide so-called misleading information. They cannot be separated, since only the whole execution time is measured. Using equations (4) and (5), the probability of correct information can be computed by

$$P_{\text{correct}}(a) = \sum_{n=\min(a,1)}^{a} P_{N|A}(n \mid a) P_A(a) \quad , \tag{6}$$

and the probability of misleading information by

$$P_{\text{misleading}}(a) = P_A(a) - P_{\text{correct}}(a) = \begin{cases} P_{N|A}(0 \mid a) P_A(a) & a \neq 0 \\ 0 & a = 0 \end{cases} . \tag{7}$$

$a = 0$ is special because in that case, the bytes p_i in the plaintext cannot be the respective key bytes k_i, because then $\texttt{InvSubBytes}(s_i) \neq 0$ in (3), i.e. the information is correct all the time. See Fig. 1 for a plot of the probability of correct and misleading information in relation to a.

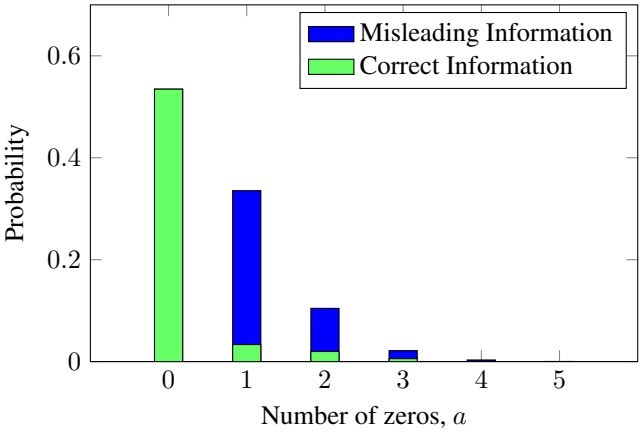

Figure 1: Probabilities of correct and misleading information in relation to the number of zeros, a. They are calculated using (6) and (7).

In order to combine the information acquired with the N individual measurements, a maximum likelihood (ML) estimator with the likelihood function L_{k_i} has been used. In order to reduce the arithmetic range during the computation, the log-likelihood function using the natural logarithm $\ln(\cdot)$ has been taken and is computed as follows

$$\ln\left(L_{k_i}(j)\right) = \ln\left(\sqrt[N]{\prod_{x=1}^{N} P_{k_i,x}(j)}\right) = 1/N \sum_{x=1}^{N} \ln\left(P_{k_i,x}(j)\right) \quad , \\ \forall i \in \{0,\ldots,15\}, \;\; j \in \{0,\ldots,255\} \quad , \tag{8}$$

where $P_{k_i,x}(j)$ is the probability that key byte $k_i = j$ in measurement x. A normalization by N has been added. The ML estimator then is

$$\hat{k}_i = \underset{j \in \{0,\ldots,255\}}{\arg\max} \; \{\ln L_{k_i}(j)\} \quad . \tag{9}$$

In the implementation, the likelihoods are stored in a likelihood matrix $\boldsymbol{V} \in \mathbb{R}^{16\times 256}$, where $v_{i+1,j+1}$ stores the likelihood for $k_i = j$. For each decryption $x \in \{1,\ldots,N\}$,

the likelihood matrix $\boldsymbol{V}$, which is initialized with zeros, is updated by adding the "vote" $\ln\left(P_{k_i,x}(j)\right)$ to the entry $v_{i+1,j+1}$. That way, after summing up the votes for all measurements, $v_{i+1,j+1} = N \ln\left(P_{k_i}(j)\right)$, c.f. (8). As a last step, each $v_{i+1,j+1}$ has to be replaced by $\exp(v_{i+1,j+1}/N)$ such that $\boldsymbol{V}$ stores the log-likelihood at the end of the estimation.

If the timing information is not considered, the probability that a single byte of the key is correct is $1/256$, because there are 256 possibilities, as usual. But if exactly one zero occurred, the probability is $1/160$ since there we assume that the zeros are distributed uniformly over the 160 inversions. Consequently, for a zeros the probability is $a/160$ and the probability for each of the other possible values is $(160 - a)/(160 \cdot 255)$, again since we assume a uniform distribution. Taking the natural logarithm leads to the voting table in Table 1 with one exception: if no zero occurred, the probability that the current bytes in the plaintext are the respective key bytes is zero. The likelihood of those bytes then is also zero, i.e. in practice a single measurement error can prevent finding the correct key, c.f. also Sect. 4. Therefore in that case, the probability for the p_i has been increased to $1/10000$ and the probability for the other values has been decreased to $(1-1/10000)/255$ instead of $1/255$.

After a certain number of measurements, there should only be one entry per key byte in the matrix which has a significantly larger vote than the other bytes, and these entries form the correct key.

3.2 Attack simulation

To test and verify the attack's theory, a simulator which is able to emulate the behavior of the AES implementation on the microcontroller was implemented in C++. If a zero is processed on the microcontroller, the algorithm skips several hundreds of cycles, which is easily measurable. The simulator therefore observes the input of `GF2_8InvMult` and counts the number of zeros, a, during decryption. Random input data is decrypted until the desired number of decryptions has been simulated and then the attack is performed. This enables the simulation of thousands of measurements in some seconds without the need for the hardware setup.

The success of the attack is highly dependent on statistical processes and so there is no fixed number of required measurements for which the attack will always be successful. The success ratio was estimated by the simulation of 4000 independent attacks, in which the number of measurements was increased in small steps from 0 to 6000. The result is depicted in Fig. 2. For less than 2000 measurements, there is almost no chance to find the correct key. The next 1000 additional measurements show the highest effect. At around 2750 measurements, the success ratio is about $50\,\%$ and at 3000, the success ratio is about $66\,\%$. It reaches $90\,\%$ at around 3550, $99\,\%$ at 4600 and finally approaches $100\,\%$. The total time needed for such an attack is only several minutes. Most of the time is required for taking the measurements. The attack itself can be computed in a few seconds on a standard computer. Eventually the attack requires neither very special equipment, nor excessive computational power and therefore can easily be performed.

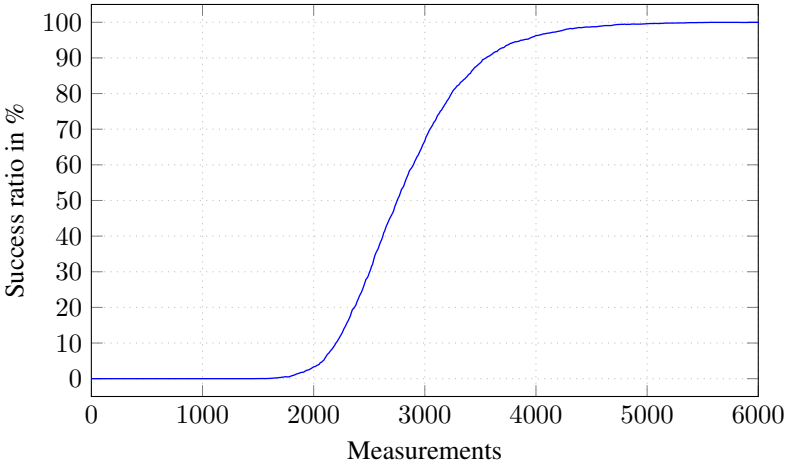

Figure 2: Success ratio in relation to the number of measurements. The curve was determined using a simulation model of the attack and executing it 4000 times with 0 to 6000 measurements each.

4 Practical implementation of the attack

The alternative implementation of the AES algorithm was loaded into the smart-card emulator, c.f. Sect. 1, so that the implementation could be tested in a real environment. The emulator was inserted into a smart-card reader, which was connected to a computer. An inexpensive USB oscilloscope was used to measure the timing, which was forwarded to the computer. The setup is depicted in Fig. 3.

The execution time is measurable in various ways, e.g. through power consumption, inter-

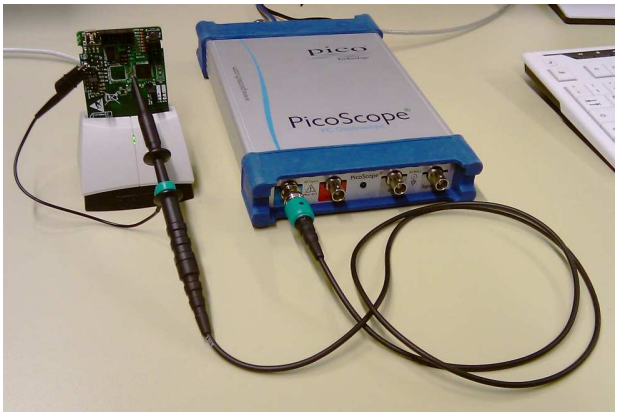

Figure 3: Measurement setup for timing attacks. The smart-card emulator is inserted into the card reader. A probe of the oscilloscope is connected to the microcontroller in the smart-card emulator. The attack is controlled by the attached PC.

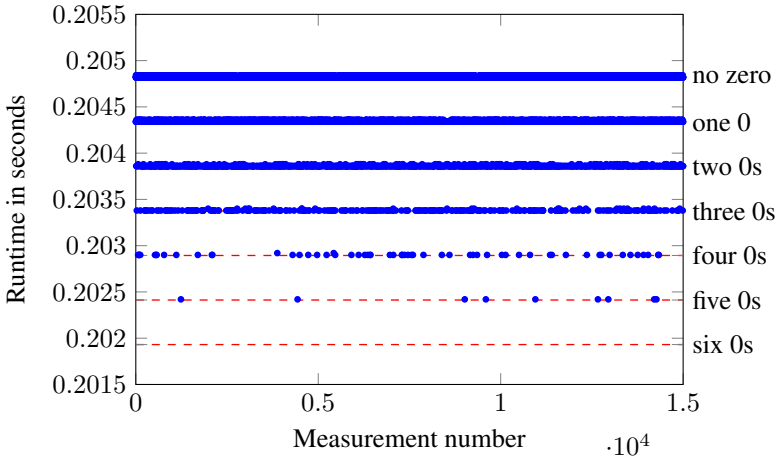

Figure 4: Input time distribution (dots) and expected run times (dashed lines). The number of points in each line complies with the predicted distribution. The dashed lines in the center of each group were automatically determined by the attack script.

face communication or dedicated pins, which indicate ongoing decryption. It is assumed to be trivial for an attacker to obtain this information through these or other methods. In our test, a microcontroller's pin was used, since it is the simplest solution and is sufficient for testing. The oscilloscope is able to determine the execution time very accurately, down to single clock cycles. The timing deviations are in the range of $0.5\,\text{ms}$ per zero at a microcontroller clock frequency of $5\,\text{MHz}$. Therefore a sample rate as low as $10\,\text{kS/s}$ is enough for these purposes and has the advantage of limiting the amount of measured data, making it easier to process. Both, the timing information and the plaintext are stored and preprocessed by a MATLAB script. Next, the number of zeros has to be estimated. The algorithm goes through the timing data and searches for the execution time, which was exceeded by more than $50\,\%$ of all measurements. The same is done for $83\,\%$. These are the cases, including some safety margin, where no zeros and one zero occurred. It gives a very good estimate for the time-saving due to each zero. The execution time for two, three and more zeros is derived from the result. There are no fixed timing values, the information is extracted by only using the expected execution time distribution. This brings

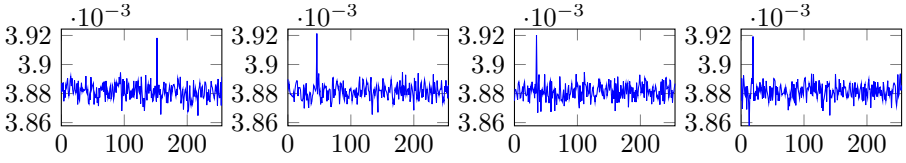

Figure 5: 4 rows of the likelihood matrix $\boldsymbol{V}$, representing the log-likelihood for each key byte. The other 12 rows look similar, only the peak position changes. The x-axes show their value, the y-axes their log-likelihood values. Each subplot shows exactly one significantly higher positive peak, which forms the correct estimated key after 15000 measurements.

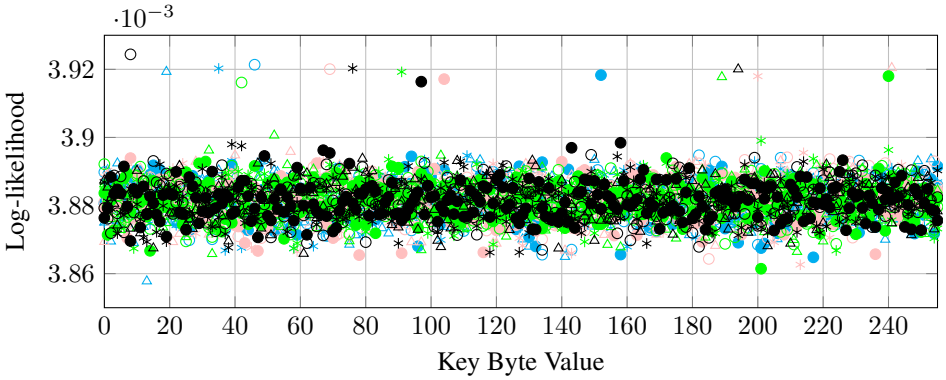

Figure 6: The full likelihood matrix $\boldsymbol{V}$. Each symbol type and shading depicts a different key byte. The incorrect estimates are concentrated in the lower plot region, the 16 correct estimates are in the upper region. Both groups separate with an increasing number of measurements. For this plot, 15000 measurements were executed.

the advantage that the script works with any microcontroller, compiler and clock speed, as long as the differences are measurable.

In order to demonstrate the performance of the estimation methods, a large measurement with 15,000 decryptions was performed. The methods work well and the result can be seen in Fig. 4. Each measurement is represented by a dot and the extracted expected execution time is shown by the dashed lines. According to the simulation in Subsect. 3.2, they contain more than enough information to extract the key. The number of zeros from the timing measurement is taken and the likelihood matrix $\boldsymbol{V}$ is computed as described in Subsect. 3.1 and plotted in two different ways in Fig. 5 and 6. One value per row in $\boldsymbol{V}$, i.e. one for each key byte, is much higher than all other 255 values in this row, as depicted in Fig. 5. It can be clearly seen that after a high number of measurements, two groups have formed in Fig. 6. The high-density area around $3.88 \cdot 10^{-3}$ contains all incorrect key byte values. In contrast to that, the 16 correct key bytes have a higher log-likelihood value of around $3.92 \cdot 10^{-3}$. At the start of the estimation, all likelihoods are the same. As the number of processed measurements increases, the correct key bytes move upwards, while the incorrect ones move downwards, on average. Finally, after a sufficient number of measurements, the correct and incorrect bytes are separated from each other and the key can be extracted.

5 Evaluation and Prospects

5.1 Countermeasures, Attack Variations and Influence on DPA Attacks

In order to defeat the timing attack, the execution time of `GF2_8InvMult` must be constant. This can be solved by modifying `GF2_8InvMult` in such a way that it does not

immediately return zero and instead uses random data whose result is discarded. The algorithm might also be replaced by a more sophisticated one, which has constant run time, but this might diminish the savings in code size.

The attack can also be used on the AES encryption, where the initial and first round are attacked. The input plaintext has to be used instead of the output. The remaining part of the attack works the same way as for the decryption.

Another variation would be a chosen plaintext attack on the encryption or decryption. If such an attack is possible, the attacker can actively guess key bytes. The number of measurements can be reduced with this method, but it requires a more sophisticated attack implementation or the manual input of data.

The implementation was also tested with a DPA attack, implemented as described in [MOP07]. It was expected that the non-constant run time leads to an increased effort for an attacker, due to trace misalignment. Nevertheless, the method was not believed to be a very secure protection against such attacks. These statements were verified experimentally and confirmed the assumptions. For normal and unprotected static S-boxes, the number of required traces was around 400. With the dynamic S-boxes and no trace alignment, even more than 1000 traces were not sufficient. After alignment of the traces, the attack was successful with around 500 traces. This shows that the effort for an DPA is increased, but the method does not provide sufficient protection.

5.2 Conclusion

We have presented a novel timing attack on our implementation of the AES decryption algorithm. As demonstrated, an attacker can find out the secret key in a negligible amount of time with an inexpensive USB oscilloscope and a common PC. Thus, the attack can be performed with very low costs. It has to be made sure that a data-dependent execution time like the one in our implementation is not even present in systems with low security requirements. But security relevant aspects such as a data-dependent execution time are missed easily. The attack shows that the way of implementing the S-boxes is critical for a secure AES implementation. But since most implementations use fixed tables, they are immune to the attack. Nevertheless, the concepts of our timing attack are useful for attacks on different devices, implementations or algorithms.

Acknowledgement

Thanks to Prof. Sigl and Oscar Guillen Hernandez, who offered and supervised the lab course, during which we developed the attack, for encouraging us to write the paper and for their valuable review and feedback.

References

[Ber04] D. J. Bernstein. Cache-timing attacks on AES, 2004. http://cr.yp.to/papers.html#cachetiming.

[BKR11] A. Bogdanov, D. Khovratovich, and C. Rechberger. Biclique Cryptanalysis of the Full AES. Cryptology ePrint Archive, Report 2011/449, 2011. http://eprint.iacr.org/.

[DR98] J. Daemen and V. Rijmen. AES Proposal: Rijndael, 1998.

[KJJ99] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M. Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer Berlin Heidelberg, 1999.

[Koc96] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In N. Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer Berlin Heidelberg, 1996.

[KS09] E. Käsper and P. Schwabe. Faster and Timing-Attack Resistant AES-GCM. In C. Clavier and K. Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, 2009.

[LN94] R. Lidl and H. Niederreiter. *Introduction to Finite Fields and Their Applications*. Cambridge University Press, 1994.

[MOP07] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks - Revealing the Secrets of Smart Cards*. Springer US, 2007.

[Nat01] National Institute of Standards and Technology. FIPS 197 - Advanced Encryption Standard, 2001.

[OST05] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and Countermeasures: the Case of AES. Cryptology ePrint Archive, Report 2005/271, 2005. http://eprint.iacr.org/.

A Implementation of `GF2_8InvMult`

`GF2_8InvMult`, shown in Listing 1, computes the multiplicative inverse v' of v over $\mathrm{GF}(2^8)$, i.e. the solution to $vv' = 1$. It has been implemented using 8-bit operations only, as an 8-bit controller has been used. `uint8_t` denotes an 8-bit unsigned integer type.

If $v =$ 0x00, 0x00 is returned according to [Nat01], c.f. lines 10 and 11. The idea of the algorithm is to find a solution by expressing $vv' = 1$ as a sum, as follows:
$1 = vv' = g^{\log_g(vv')} = g^{p+p'}$, where $p = \log_g v,\ p' = \log_g v'$ and g is a base.

The rather simple and space-saving algorithm is composed of three steps. The first is to calculate $p = \log_g(v)$. There exist bases g on $\mathrm{GF}(2^8)$, for which a p can be found for every $v \neq 0$, i.e. its powers result in every element of the field, except 0. Such a g is called primitive element [LN94]. The brute-force search starts with $p = 0$, calculates g^p and increases p until $g^p = v$, c.f. the loop in lines 12 to 16. In each iteration of the loop g^p is multiplied by g. As the primitive element is of the order 255, the search will terminate after at most 255 iterations.

```
1  inline uint8_t Mult3GF2_8(uint8_t gP) /* multiply by 0x03 (x + 1) */
2  {
3       uint8_t prod = gP << 1; /* multiply by 0x02 (x) */
4       if (gP & 0x80) /* reduce modulo (x^8 + x^4 + x^3 + x + 1) */
5            prod ^= 0x1b;
6       return gP ^ prod; /* calc. result (x) + (1) */
7  }
8  uint8_t GF2_8InvMult(uint8_t v)
9  {
10      if (v == 0) /* multiplicative inverse of 0 is defined as 0 */
11           return 0;
12      uint8_t p = 0, gP = 1;
13      while (gP != v) { /* calculate 0x03^p = v, solve for p */
14           gP = Mult3GF2_8(gP);
15           p++;
16      } /* calculate the multiplicative inverse gP = 0x03^(255-p) */
17      for (gP = 1; p < 255; p++)
18           gP = Mult3GF2_8(gP);
19      return gP;
20 }
```

Listing 1: Source Code of `GF2_8InvMult` for the calculation of the multiplicative inverse

The second step is to calculate p'. g^0 has to equal $g^{p+p'}$ in $\mathrm{GF}(2^8)$. As there are only 255 different elements in the powers of g, any sum of p and p' which is a multiple of 0xFF results in the same value as 0x00. In order to find p', $0x00 = (p + p') \mod 0xFF$ should be solved by $p' = 0xFF - p$. In the code, this step has been combined with the third step, the calculation of the power $v' = g^{p'}$, c.f. lines 16 to 18. Calculating p' is omitted, instead the correct number of successive multiplications by g to compute v', $255 - p$ are executed anyway by the careful design of the loop. The result v', the inverse of v, is then returned.

Please note that the elements of $\mathrm{GF}(2^8)$ are polynomials which have an equivalent notation as an 8-bit hexadecimal number, e.g. $x + 1 = 0x03$. Choosing the primitive element $g = x + 1 = 0x03$ is beneficial, because the multiplication by it can be implemented efficiently. In `Mult3GF2_8` (lines 1 to 7), it is split into multiplying by 1, which is redundant, and by x. For this multiplication, `gP` is shifted left by one bit. If the MSB of `gP` is set, the result of the multiplication cannot be represented in $\mathrm{GF}(2^8)$ and the result still needs to be reduced modulo the polynomial $x^8 + x^4 + x^3 + x + 1$, which in this case can be simplified to subtracting this polynomial, which is equivalent to xoring the result with 0x1b [Nat01]. In the last step, `gP` and the product are added.

Regarding the security of `GF2_8InvMult`, it has to be analyzed whether the search and its intentionally simple implementation leaks any critical information. The number of iterations of the first loop depends on the input data, but together with the second loop, the total number of iterations is 255. Nevertheless, there is some minor run time difference due to the if statement, and the compiler may also implement the second loop differently than the first one. The source of the largest timing variation can easily be seen in lines 10 and 11. If the input value is zero, the function returns zero immediately and does not execute the loops, which makes the implementation vulnerable to a timing attack, as shown in Sect. 3.