A Continuous Fault Countermeasure for AES Providing a Constant Error Detection Rate

Marcel Medwed

Graz University of Technology, Institute for Applied Information Processing and Communications, Inffeldgasse 16a, 8010 Graz, Austria
Marcel.Medwed@iaik.tugraz.at

Abstract. Many implementations of cryptographic algorithms have shown to be susceptible to fault attacks. For some of them, countermeasures against specific fault models have been proposed. However, for symmetric algorithms like AES, the main focus of available countermeasures lies on performance so that their achieved error detection rates are rather low or not determinable at all. Even worse, those error detection rates only apply to specific parts of the cipher. In this paper we present a way to achieve a constantly higher error detection rate throughout the whole algorithm while assuming a much stronger adversary model than in previous papers. Furthermore, we propose solutions for two very important, unsolved questions: First, how to do secure and efficient table lookups in redundant algebras. Second, how to implement secure correctness checks to verify the result in a scenario where the adversary can manipulate comparisons. Our paper is therefore the first one to construct a sound and continuous AES fault countermeasure with an attacker-independent minimum error detection rate.

Keywords: Fault attacks, countermeasure, AES, EAN+B codes, redundant table lookups, secure correctness checks.

1 Introduction

In 1997, Boneh, Demillo, and Lipton presented a new type of cryptanalysis, namely fault analysis [7]. The principal idea behind these attacks is to manipulate the computation and evaluate the faulty behavior of the device in order to learn something about the used secret. Whereas their attack targeted asymmetric algorithms, Biham and Shamir followed up shortly afterwards with an attack on DES [4]. Since then, many attacks [6, 8, 11, 12, 19–21] and countermeasures [13–15, 17] have been introduced.

The way in which an attacker is able to manipulate the computation is called the fault model. All these attacks and countermeasures assume specific fault models. However, especially for symmetric algorithms the fault model assumed for countermeasures often differs from the one used for attacks. One such example is the countermeasure proposed by Karpovsky et al. in [14]. In their paper, each AES round is protected separately. Exactly this characteristic of the countermeasure makes it vulnerable to attacks. This is because there is no protection between the rounds. Thus, the architecture meets exactly the requirements for the attack proposed by Dusart et al. in [11]. In order to overcome

the problem, where a countermeasure is only secure under specific fault model assumptions, we assume a much stronger adversary and secure the AES algorithm throughout the whole computation.

Apart from the fundamental question of the assumed fault model, there arises also the question of how to check the correctness of the result returned by a protected algorithm. Since instructions can be skipped [21], checks can be manipulated as well. Therefore, their implementation is at least as crucial as the implementation of the countermeasure itself. The best solution is to make such checks obsolete by breaking the correlation between the fault and the output. An example for such an approach is presented in [5]. Unfortunately, this is not always possible. Hence, there exists the necessity for secure checks. We present a probabilistic approach to solve this problem.

Furthermore, for AES, a major challenge for fault-countermeasure design is the non-linearity of the S-Box. This is because non-linear functions are much harder to protect than linear ones. The problem is further aggravated, if it is desired to use the same protection mechanism for both parts in order to achieve a continuous protection. All previously proposed countermeasures treat the linear and the non-linear part separately and therefore leave vulnerabilities. We show how to do secure table lookups in redundant algebras and hence provide a solution to this problem. Finally, we put these elements together to implement a reasonably protected and sound AES implementation.

The remainder of the paper is organized as follows: Section 2 describes the used adversary model, followed by a short review of the AES algorithm in Section 3. Section 4 discusses extended AN+B codes, how to apply them to AES, and how to implement secure table lookups with such codes. After proposing methods to implement secure correctness checks in Section 5, we assemble our AES implementation and investigate its security properties in Section 6. We briefly analyze the countermeasure's influence on side-channel analysis in Section 7. A performance analysis is done in Section 8 before we draw conclusions in Section 9.

2 Adversary Model

Usually, countermeasures are only shown to be secure under a specific fault model. For our research it was important not to restrict the capabilities of the attacker too much. To classify the attacker we stick to the adversary classes introduced in [1] which can be briefly summarized as follows: the clever outsider (class I), the knowledgeable insider (class II) and the funded organization (class III). It is widely believed and reasonable to assume that chip content cannot be secured against a class III attacker [2]. This is because such an attacker can, by investing large amounts of money and time, remove or circumvent all hardware countermeasures and afterwards probe all of the chip's internals. Therefore, we assume a very strong class II adversary. We allow such an attacker to inject the following faults:

- **Bit-set/flip fault:** The adversary can set a specific bit within the memory or registers either to zero or to one. Alternatively, a bit can be flipped. The position and the moment in time can be chosen by the attacker.
- **Random byte fault:** This fault is similar to the one above, except that the error cannot be chosen as precisely. In other words, the attacker can add a random value

$\epsilon \in [1, 255]$ to a byte in the memory or to a register, whereas the addition is an exclusive-or.

– **Skip instructions:** These faults are quite common in real life and have been used to attack pay-TV smart cards [3]. It has also been shown that they can be used to skip comparisons in order to bypass correctness checks and therefore defeat countermeasures [16].

Furthermore, an adversary is allowed to induce all the above faults several times during a single run of the algorithm. Each attempt to induce a fault succeeds with a probability of one. Finally, the attacker is allowed to induce the first two fault types permanently. This means for instance that wires can be cut. As a result, the same error is present during all following runs of the algorithm. Such errors usually defeat redundant rounds.

3 AES

In 2000, a special version of the Rijndael algorithm was chosen to serve as the new US standard for symmetric encryption. This Advanced Encryption Standard (AES) is a block cipher which operates on a state of 128 bits and works with key sizes of 128, 192 or 256 bits. The state is presented by a 4x4 byte matrix, where each byte is an element of $GF(2^8)$. Depending on the key size, the number of rounds can be either 10, 12, or 14. We only look at the 128-bit version, consisting of 10 rounds. At the beginning of the algorithm, the key is expanded into 11 round keys, each of them consisting of 16 bytes. For details about the key scheduling refer to [18]. The cipher itself consists of one AddRoundKey operation at the beginning, followed by 9 AES rounds and a final round. Each of the 9 rounds is composed of four transformations, namely SubBytes, ShiftRows, MixColumns and AddRoundKey. The final round differs from the others by a missing MixColumns operation. To describe the four AES transformations we use the following notation:

– S_i denotes the ith byte of the state for $0 \leq i \leq 15$.
– $K_{i,k}$ denotes the ith byte of the kth round key for $0 \leq k \leq 10$.
– C_l denotes the polynomial modulo $y^4 + 1$ which takes the four elements of the lth column (in descending order) as coefficients, with $0 \leq l \leq 3$, i.e.

$$C_l = S_{12+l} \cdot y^3 + S_{8+l} \cdot y^2 + S_{4+l} \cdot y + S_l.$$

– R_m denotes the polynomial modulo $y^4 + 1$ which takes the four elements of the mth row (in ascending order) as coefficients, with $0 \leq m \leq 3$, i.e.

$$S_{4m} \cdot y^3 + S_{4m+1} \cdot y^2 + S_{4m+2} \cdot y + S_{4m+3}.$$

– $+$ denotes the exclusive-or operation since the used field has characteristic 2.

Using this notation the four round operations of round k can be described as following:

AddRoundKey

$$S_i = S_i + K_{i,k}$$

SubBytes

$$S_i = A * (S_i^{-1} \pmod{x^8 + x^4 + x^3 + x + 1)) + d$$

After the inversion each state byte is treated as a bit vector. A denotes a fixed 8×8 bit matrix and d a constant 8×1 bit vector. The matrix multiplication and the vector addition are done over $GF(2)$. If a state byte has the value 0, the inversion is skipped.

ShiftRows

$$R_m = R_m * y^m \pmod{y^4 + 1}$$

MixColumns

$$C_l = C_l * (3y^3 + y^2 + y + 2) \pmod{y^4 + 1}$$

4 Extended AN+B codes

Extended AN+B codes have been introduced in [17] as a generic countermeasure against fault attacks. In this section we review this method to calculate with redundancy. The approach is similar to common ring extension methods. However, the main difference is that the result modulo the extending modulus is data independent and precomputed. Hence, its computation cannot be manipulated since it happens during the key generation.

The basic principle behind extended AN+B codes is to take a suitable algebra (e.g. RSA ring, ECC base field, or AES field) and extend it by a second one. We refer to the first one as the data algebra $\mathbb{F}_D$ and to the second as the check algebra $\mathbb{F}_C$. Using the direct product of the two algebras, they can be rewritten as Cartesian product ($\mathbb{F}_D \times \mathbb{F}_C$). The algebraic operations are further defined componentwise. A tuple of two elements $x_D \in \mathbb{F}_D, x_C \in \mathbb{F}_C$ is written as column vector. Therefore, applying the algorithm $f()$ on such a tuple results in

$$f\left(\begin{pmatrix} x_D \\ x_C \end{pmatrix}\right) = \begin{pmatrix} f(x_D) \\ f(x_C) \end{pmatrix}.$$

The aim of this method is to enlarge the number of possible elements, while keeping the number of valid elements constant. Instead of $|\mathbb{F}_D|$ elements the new algebra contains now $|(\mathbb{F}_D \times \mathbb{F}_C)|$ possible elements. However, to keep the number of valid elements at $|\mathbb{F}_D|$, the check algebra is only allowed to hold a specific value at the end of the algorithm. This value has to be precomputed. Since $f(x_D)$ and $f(x_C)$ are independent of each other, this is always possible for non-probabilistic algorithms.

However, in order to make sure that a correct x_C implies a correct x_D with a certain probability, the same algorithm has to be applied to both elements. For this purpose, the algebras must not only be combined theoretically, but also physically within the device. This is done by applying the Chinese remainder theorem (CRT). From a structural point of view nothing changes since

$$\begin{pmatrix} \mathbb{F}_D \\ \mathbb{F}_C \end{pmatrix} \cong \mathrm{CRT}(\mathbb{F}_D, \mathbb{F}_C)$$

but the representation of the data within the microcontroller changes in an interesting way: Assume that both, x_D and x_C, occupy 8 bits in memory, hence 16 bits together, before running the CRT algorithm. Then afterwards, x_D uses 16 bit as well as x_C. However, they are orthogonally encoded and therefore they can be added and stored at the same place in memory without loss of information. Thus, calculations are still applied componentwise, but it is guaranteed that each operation is applied to both components. Furthermore, alterations of the memory do affect both components with maximal probability of $1 - 1/|\mathbb{F}_C|$.

In summary, extended AN+B codes have three properties which are interesting for our purpose. First, they do not only secure operands throughout addition and multiplication, but also allow only a specific function (i.e. a specific key) to be applied to the encoded operands. This is because if the key or the instruction sequence (e.g., by skipping instructions) changes, also the applied function $f()$ changes to $\tilde{f}()$. At the end of the algorithm, it can be verified that $f(x_C) \neq \tilde{f}(x_C)$. Second, if several variables are secured and each of them is assigned a different signature x_C, then the variables are not interchangeable anymore. This is vital for a code in order to secure AES. The third property which allows the construction of redundant table lookups is described in Section 4.2.

4.1 EAN+B Codes Suitable for AES

The previous section described the principle of extended AN+B codes and how they are constructed. This section deals with the question of how to choose the algebras for AES. We decided to add another 8 bits to each state byte. A redundancy of 8 bits is the minimum needed, in order to continuously protect the algorithm.

A straightforward approach to add these 8 bits would be to take the fields $\mathbb{F}_D$ and $\mathbb{F}_C$ as they are and construct a ring consisting of polynomials of degree smaller 16. Although this is possible from a mathematical point of view, it is not advisable from a computational point of view. The approach would penalize the most costly operation within our AES implementation, the polynomial multiplication, for basically two reasons: First, usually, microcontrollers do not feature a GF(2^m) multiplier. Hence, the multiplication of two elements has to be done by shift operations and conditional XOR operations. However, this is too costly in terms of execution time. Second, a fast approach would be too expensive in terms of memory: Multiplication can also be done using logarithm tables and in fact this approach is well suited for GF(2^8) where the logarithm table needs only 256 entries. For our ring, consisting of polynomials of degree smaller 16, on the other hand, we would need 2^{16} entries. Therefore, we have to construct a different ring, which allows a trade-off between execution time and memory. A ring $\mathbb{U}$, consisting of polynomials of degree smaller 2, with coefficients in GF(2^8) allows the use of logarithm tables with 256 entries, although $|\mathbb{U}| = 2^{16}$. The bijective mapping we are looking for in order to map the (x_D, x_C) tuples into $\mathbb{U}$ is the following:

$$\phi : \begin{pmatrix} \mathbb{F}_D \\ \mathbb{F}_C \end{pmatrix} \rightarrow \mathbb{U} := (\mathbb{F}_2[x]/P)[y]/y^2 + c_1 y + c_0$$

with the irreducible polynomial P defining the base field $\in GF(2^8)$ and the coefficients c_1 and c_0 being elements of this base field. This mapping is applied to the operands

before the AES calculation. After running the algorithm its inverse is used to transform them back.

To construct the ring $\mathbb{U}$ we use the two fields $\mathbb{F}_{U1} := (\mathbb{F}_2[x]/P)[y]/y + a_1$ and $\mathbb{F}_{U2} := (\mathbb{F}_2[x]/P)[y]/y + a_2$ with $a_1 \neq a_2$. Hence, the first step is to use ϕ_1 to map the elements of $\mathbb{F}_D$ to $\mathbb{F}_{U1}$ and ϕ_2 to map the elements of $\mathbb{F}_C$ to $\mathbb{F}_{U2}$. However, if the base field is chosen to be the AES field, these mappings do not have to be implemented, because the representation of the elements in the microcontroller does not change. By multiplying the moduli of $\mathbb{F}_{U1}$ and $\mathbb{F}_{U2}$ it is now possible to construct the desired two-term ring $\mathbb{U}$ with $c_1 = (a_1 + a_2)$ and $c_0 = a_1 a_2$. All left to do now, is to calculate the two idempotent elements for the CRT, i_1 and i_2, as follows: With $p_1 = y + a_1$ and $p_2 = y + a_2$ we get

$$i_1 = p_2(p_2^{-1} \pmod{p_1}) \pmod{p_1 p_2} \tag{1}$$

$$i_2 = p_1(p_1^{-1} \pmod{p_2}) \pmod{p_1 p_2}. \tag{2}$$

Finally the mapping ϕ and its inverse can be defined as:

$$\phi : (\mathbb{F}_D \times \mathbb{F}_C) \rightarrow \mathbb{U} \tag{3}$$

$$\phi_1(x_D) \cdot i_1 + \phi_2(x_C) \cdot i_2 \tag{4}$$

$$\phi^{-1} : \mathbb{U} \rightarrow (\mathbb{F}_D \times \mathbb{F}_C) \tag{5}$$

$$\begin{pmatrix} \phi_1^{-1}(u \pmod{p_1}) \\ \phi_2^{-1}(u \pmod{p_2}) \end{pmatrix} \tag{6}$$

As stated above, if the irreducible polynomial P equals the AES polynomial, then the mappings ϕ_1 and ϕ_2 are only of theoretic nature. Using $\phi(x_D, x_C)$, the bytes of the plaintext and those of the signature can be transformed pairwise into elements of $\mathbb{U}$. After the AES calculation they are transformed back using $\phi^{-1}(u)$, where u is an element of $\mathbb{U}$.

4.2 Redundant Table Lookups

A question that remains is how to implement the SubBytes operation used by AES. Since inversions are costly in software, this transformation is normally realized by using a 256-byte lookup table. However, a lookup table with 2^{16} entries is not practical for our case. Extracting x_D and using a standard lookup table (S-Box) is no option either, since this would present a vulnerability. Hence, it would be convenient to reduce the problem to an efficient, yet redundant, table lookup.

By fixing the signature to a known value before a lookup, we can reduce the information held by the 16-bit value to 8 bits. Fortunately, there is also a fast way to get hold of the encoded information: For both of the polynomial's coefficients, there exists a bijective mapping to $\mathbb{F}_D$ (see Appendix A.1), if both coefficients of the idempotent i_1 are co-prime to P. This is given since P is irreducible. The actual mapping is for free, since an S-Box, already permuted according to this mapping, can be precomputed.

Nevertheless, both coefficients have to be used, otherwise it is as insecure as extracting x_D. The idea to protect the S-Box lookup is to take the first coefficient of the

input polynomial to locate a 16-bit S-Box value. This first result contains an error term depending on x_D. The second coefficient is used to look up the corresponding correction term. In other words, if either of the indices (coefficients) has been altered, they refer to different x_D values and the output of the S-Box is corrected to a wrong x_C. The signature before the lookup and the signature afterwards have to be different, otherwise the operation would not affect x_C and could be skipped. We denote the fixed input signature before the lookup as $x_{C_{in}}$ and to the fixed output signature after the lookup as $x_{C_{out}}$.

Although the signature x_C has to be normalized before each S-Box lookup, it is crucial that the output signature correlates with the input signature before the normalization. Otherwise the protection would not be continuous. We propose to store the difference between the actual and the normalized input signature and to add this difference again after the lookup. A complete S-Box lookup can be seen in Algorithm 1. A detailed description of the algorithm can be found in Appendix A.2.

Algorithm 1 Redundant S-Box Lookup

Require: An input value $u = u_1 y + u_0 = \mathrm{CRT}(\phi_1(x_D), \phi_2(x_C))$, the transformed input signature $m = x_{C_{in}} \cdot i_2$, the idempotent i_2, the redundant S-Box table SB, and the correction table CT.

Ensure: $u_1 y + u_0 = \mathrm{CRT}(\phi_1(\mathrm{SubBytes}(x_D)), \phi_2(x_C + x_{C_{in}} + x_{C_{out}}))$

1: $t \leftarrow u \cdot i_2$
2: $d \leftarrow t + m$
3: $u' \leftarrow u + d$
4: $u \leftarrow SB[u'_1] + CT[u'_0] + d$
5: **return** u

5 Secure Correctness checks

Amongst the many countermeasures proposed so far, none defines the final checks themselves. In this section we propose methods which leave the adversary with only a probabilistic chance to succeed in bypassing these checks after inducing an error. Assuming devices which deactivate themselves after a certain number of detected faults, the aim is to prevent the attacker from gathering useful information from training devices. In other words the attacker must not be able to improve his knowledge about how to bypass the check with the help of training devices. The following three building blocks are used in order to achieve this:

Time randomization: Here, the moment in time, when the checks happen, depends on the device itself. Hence, there are n time slots, when the check of byte i can take place and it does in time slot j. This permutation of the checks depends on the device and does not need to change for every run of the algorithm. The attacker succeeds with a probability of $1/n$ to mount the attack at the right moment in time. In the case of AES, n would be 16, since there are 16 signature bytes x_C. Apart from the fact that this error

detection probability is smaller than the one of the code itself, one could argue that an instruction skipping attack can be mounted on all time slots. Therefore a randomization of the branches is required as well.

Branch randomization: Randomizing the branches basically means that the adversary does not know, which branch to choose, which instruction to skip or to which value a flag must be set. This technique decreases the adversary's probability only by 50%. Nevertheless, it is vital for the whole concept. The implementation of these randomized branches is crucial as well. This is because an attacker, using simple power analysis, can observe the power consumption of a correct run of the algorithm and recover the randomization. Therefore, we propose to first evaluate a branch condition (i.e. $con \leftarrow x_C \neq f(x_{C_{init}})$ or $con \leftarrow x_C = f(x_{C_{init}})$) and then use this condition to compute a destination address for the code to be executed (i.e. $dest \leftarrow con * delKey + not(con) * output$ or $dest \leftarrow not(con) * delKey + con * output$).

Check doubling: Another technique is to run the checks t times. Thus, if a single run of the checks had probability p, then t runs with distinct randomization guarantee a probability of p^t. This method is suitable to ensure that the checks are not less secure than the remainder of the implementation and hence decrease the overall security.

5.1 Integration

By combining the time randomization with the branch randomization the attacker is left with a chance of $1/32$. If the attacker needs to manipulate the checks for more than one byte, the success probability decreases. However, assuming the probability with $1/32$, the check still presents the weakest link in the whole system. Therefore, we repeat the checks once with different randomization parameters and reduce the attacker's chance to $1/1024$.

The randomization of the checking mechanism itself does not need to change on every run of the algorithm. It is enough if it changes per device. Therefore, the randomized instruction sequence can be generated together with the key.

6 Implementation and Security

This section investigates the security of the whole system, considering several attack scenarios. The first threat we take a closer look at, is the manipulation of data in memory. By equipping every state byte with a signature, we can provide reasonable security against such threats. Recalling the scenarios from Section 2, we see that for example single-bit errors are not possible, since the adversary needs to add a multiple of the idempotent in order to succeed. This also holds for byte errors, for the same reason. In fact, all additive errors, which are not larger than the added redundancy cannot lead to success. Hence, if the adversary wants to manipulate a data value in the memory, he can only succeed with a probability of $1/256$ by attacking two bytes. First, one byte is altered with probability 1. Afterwards, there is only one possibility left for the second byte in order to preserve the signature. Therefore, since a signature is added to each

byte at the beginning and is only removed at the end of the cipher, the data integrity in memory is preserved throughout the computation with probability $1/256$.

However, if every byte is protected with the same signature, they are interchangeable. Such an implementation would be susceptible to index manipulation. For example, a wrong state byte could be loaded during an operation. This would allow random byte errors like the ones required for the attack described in [11]. To prevent such an attack, different signatures have to be assigned to every byte of the state. The x_C values can be seen as random values throughout the algorithm. Thus, it can happen, that two take the same value at a certain point in time and become interchangeable. By applying the birthday paradox, we get a probability of $1 - \frac{256!}{256^{16}240!} \approx 0.38$ for such an event. However, the adversary still has to guess or randomly choose those indices. This hardens the task by a probability of $1/(15 * 16)$. Combining the two events results in an overall probability of $\approx 1/630$.

Furthermore, none of the operations must be negligible for the computation of the final signature. However, this alone does not suffice. Every operation has to affect the signature in a way, that even two or more skipped operations are discovered at the end. That is, no instruction presents a trivial inverse of another one. Of course since the signatures are considered random throughout the algorithm, such events can occur. An example would be the following: AddRoundKey alters the signature from x'_C to $x'_C + k_C$. If $k_C = x_{C_{out}} + x_{C_{in}}$, the attacker would succeed in skipping the AddRoundKey and the SubBytes operation. However, such an event occurs with probability $1/256$ and furthermore if such operations are not adjacent, it becomes more difficult, since the error spreads all over the state. This value must be multiplied with the probability of the attacker to find out at which position this event occurs. If we add a random signature (per key) to the plaintext, we can observe that no attack can succeed with a probability larger than $1/256$ throughout the whole cipher.

Finally, we show that this holds as well for the redundant S-Box lookup, as described in Algorithm 1. Taking a look at the algorithm, we see that every line or instruction within the line directly or indirectly changes the signature. Furthermore, no two lines present a trivial (data independent) inverse of each other. Although, as stated before, depending on the data processed, it can happen that two instruction are inverse. As for the lookup itself, u'_0 and u'_1 respectively can only be altered together with a probability of $1/256$ for the same reasons as above.

We hence can state for our countermeasure that the attacker cannot succeed at all with bit errors and random byte errors. For all other attacks the adversary succeeds with a probability of $1/256$ or less.

6.1 Comparison with Existing Countermeasures

A direct security comparison between existing countermeasures and ours is difficult since the fault model and the approach to the problem differ. We can state a maximal success probability for the attacker, whereas for time-redundant or space-redundant approaches this is not possible. Also for non-continuous approaches security figures can only be stated for parts of the algorithm, thus leaving the security of some parts unspecified. In contrast to countermeasures which assume a specific/restricted fault model, for

our approach the adversary's chances will not increase with maybe new upcoming fault injection techniques.

Software fault-countermeasures for AES normally deploy some kind of time redundancy. While this approach is cheap and straight-forward, it usually does not detect permanent and destructive faults. In contrast to such methods, our countermeasure detects permanent faults since we do not compare the results of consecutive runs. Furthermore, our countermeasures also implicitly prevents attacks on the key schedule as presented in [8].

Space-redundant hardware approaches, depending on the implementation, might be vulnerable to program flow manipulation. For instance, skipping the last round in [13] suffices to recover half of the key bits of the last round key. Other approaches like in [14] leave time windows open in which no protection is provided. Our approach has the advantage that it provides program flow integrity and continuously protects the algorithm. Furthermore, it is superior over approaches which focus on small error multiplicities like [9] since faults up to 8 bit are not possible at all.

7 Consequences for Side-Channel Attacks

Although our presented fault-attack countermeasure was never intended to counteract side-channel attacks, we investigated the possibilities. In the case of AES, the memory overhead of the employment of this countermeasure against side-channel attacks is impractical. This is because different initial signatures as well as different $x_{C_{in}}, x_{C_{out}}$ values would have to be used. These values would serve as masks for the S-Box. However, for different $x_{C_{in}}, x_{C_{out}}$ values, also different lookup tables are needed. The S-Box table of our implementation needs 512 bytes and the correction table needs 256 bytes. Hence, for every additional mask, another 768 bytes of memory are needed. Even worse, these 768 extra bytes, do not provide much extra security. If we randomly change the used mask for every block, the countermeasure can be seen as a time randomization, where a window has already been applied. Therefore, introducing n different masks reduces the DPA peak linearly by n. Hence, although this technique could be applicable in combination with other DPA countermeasures for other ciphers, it is not for AES.

8 Performance

The performance of the used code depends on the size of the added redundancy. In the case of AES, the minimum redundancy in order to continuously protect the implementation is 100%. Therefore, simple operations like AddRoundKey need twice the execution time. SubBytes on the other hand needs three single byte lookups plus eight additions and a ring multiplication. Furthermore, the ring multiplication in our algebra needs three addtions and six multiplications in $GF(2^8)$, which are performed using logarithm tables. Rijndael was not designed towards the use of such an algebra and therefore the implementation becomes costly. On an ATMega128 microcontroller roughly 90,000 cycles are needed for 10 rounds of AES. Since the two ring multiplications (one for the SubBytes and one for MixColums) alone need 54,000 of the cycles, a hardware

acceleration for this operation might help. Another possibility to speed up the implementation would be to use T-tables [10]. The use of redundant lookups would allow such a performance increasing technique, however in resource-restricted environments the use of T-tables might not be possible. We are aware of the fact, that the overhead of our implemenation is large in terms of execution time. On the other hand, the approach creates no extra hardware costs.

9 Conclusion

We presented a fault-secure AES implementation. In contrast to previous works we focused predominantly on security rather than on performance and assumed a stronger adversary. We presented methods to do secure correctness checks and showed how redundant table lookups can be implemented. Using these two building blocks together with EAN+B codes, we constructed a sound, fault-protected AES implementation. To the best of our knowledge this is the first work to continuously protect the AES algorithm with a constant error detection rate. The security evaluation shows that a standard attacker (capable of bit and word-width errors, random or not) cannot succeed at all. For any other attacker we can state an upper success rate bound of $1/256$. Furthermore, the implementation was designed in a way that an attack working for a single device is not applicable to another device. This means that the amount of devices the attacker possesses for training purposes does not increase the chances for the target device.

References

1. D. G. Abraham, G. M. Dolan, G. P. Double, and J. V. Stevens. Transaction Security System. *IBM Systems Journal*, 30(2):206–229, June 1991.
2. R. J. Anderson and M. G. Kuhn. Tamper Resistance - a Cautionary Note. In *Second Usenix Workshop on Electronic Commerce*, pages 1–11, November 1996.
3. R. J. Anderson and M. G. Kuhn. Low Cost Attacks on Tamper Resistant Devices. In M. L. et al. editor, *Security Protocols, 5th International Workshop*, volume 1361 of *Lecture Notes in Computer Science*, pages 125–136. Springer, 1997.
4. E. Biham and A. Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In B. S. K. Jr., editor, *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 1997.
5. J. Blömer, M. Otto, and J.-P. Seifert. A New CRT-RSA Algorithm Secure Against Bellcore Attacks. In S. Jajodia, V. Atluri, and T. Jaeger, editors, *Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS 2003, Washington, DC, USA, October 27-30, 2003*, pages 311–320. ACM, October 2003.
6. J. Blömer and J.-P. Seifert. Fault Based Cryptanalysis of the Advanced Encryption Standard (AES). In R. N. Wright, editor, *Financial Cryptography, 7th International Conference, FC 2003, Guadeloupe, French West Indies, January 27-30, 2003, Revised Papers*, volume 2742 of *Lecture Notes in Computer Science*, pages 162–181. Springer, January 2003.
7. D. Boneh, R. A. DeMillo, and R. J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In W. Fumy, editor, *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceedings*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997.

8. C.-N. Chen and S.-M. Yen. Differential Fault Analysis on AES Key Schedule and Some Coutnermeasures. In R. Safavi-Naini and J. Seberry, editors, *Information Security and Privacy, 8th Australasian Conference, ACISP 2003, Wollongong, Australia, July 9-11, 2003, Proceedings*, volume 2727 of *Lecture Notes in Computer Science*, pages 118–129. Springer, 2003.
9. M. Czapski and M. Nikodem. Error detection and error correction procedures for the advanced encryption standard. In P. Charpin and T. Helleseth, editors, *Designs, Codes and Cryptography*, volume 49, pages 217–232, Norwell, MA, USA, 2008. Kluwer Academic Publishers.
10. J. Daemen and V. Rijmen. *The Design of Rijndael*. Information Security and Cryptography. Springer, 2002. ISBN 3-540-42580-2.
11. P. Dusart, G. Letourneux, and O. Vivolo. Differential Fault Analysis on A.E.S. In J. Zhou, M. Yung, and Y. Han, editors, *Applied Cryptography and Network Security, First International Conference, ACNS 2003. Kunming, China, October 16-19, 2003, Proceedings*, volume 2846 of *Lecture Notes in Computer Science*, pages 293–306. Springer, October 2003.
12. C. Giraud. DFA on AES. In H. Dobbertin, V. Rijmen, and A. Sowa, editors, *Advanced Encryption Standard - AES, 4th International Conference, AES 2004, Bonn, Germany, May 10-12, 2004, Revised Selected and Invited Papers.*, volume 3373 of *Lecture Notes in Computer Science*, pages 27–41. Springer, May 2004.
13. M. Joye, P. Manet, and J.-B. Rigaud. Strengthening Hardware AES Implementations against Fault Attacks. *IET Information Security*, 1(3):106–110, Sept. 2007.
14. M. G. Karpovsky, K. J. Kulikowsk, and A. Taubin. Differential Fault Analysis Attack Resistant Architectures for the Advanced Encryption Standard. In J.-J. Quisquater, P. Paradinas, Y. Deswarte, and A. A. E. Kadam, editors, *Sixth International Conference on Smart Card Research and Advanced Applications (CARDIS '04), 23-26 August 2004, Toulouse, France*, pages 177–192. Kluwer Academic Publishers, August 2004.
15. M. G. Karpovsky, K. J. Kulikowski, and A. Taubin. Robust Protection against Fault-Injection Attacks on Smart Cards Implementing the Advanced Encryption Standard. In *2004 International Conference on Dependable Systems and Networks (DSN 2004), 28 June - 1 July 2004, Florence, Italy, Proceedings*, DSN, pages 93–101. IEEE Computer Society, 2004.
16. C. H. Kim and J.-J. Quisquater. Fault Attacks for CRT Based RSA: New Attacks, New Results, and New Countermeasures. In D. Sauveron, C. Markantonakis, A. Bilas, and J.-J. Quisquater, editors, *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems, First IFIP TC6 / WG 8.8 / WG 11.2 International Workshop, WISTP 2007, Heraklion, Crete, Greece, May 9-11, 2007, Proceedings.*, volume 4462 of *Lecture Notes in Computer Science*, pages 215–228. Springer, 2007.
17. M. Medwed and J.-M. Schmidt. A Generic Fault Countermeasure Providing Data and Program Flow Integrity. In *Fault Diagnosis and Tolerance in Cryptography, Third International Workshop, FDTC 2008, Washington DC, USA, August 10, 2008, Proceedings*. IEEE-CS Press, August 2008.
18. National Institute of Standards and Technology (NIST). FIPS-197: Advanced Encryption Standard, November 2001. Available online at `http://www.itl.nist.gov/fipspubs/`.
19. D. Peacham and B. Thomas. A DFA attack against the AES key schedule. SiVenture White Paper, October 2006.
20. J.-J. Quisquater and G. Piret. A Differential Fault Attack Technique Against SPN Structures, with Application to the AES and KHAZAD. In C. P. C. Walter, C. K. Koc, editor, *Fifth International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)*, pages 77–88. Springer-Verlag, 2003.

21. J.-M. Schmidt and C. Herbst. A Practical Fault Attack on Square and Multiply. In *Fault Diagnosis and Tolerance in Cryptography, Third International Workshop, FDTC 2008, Washington DC, USA, August 10, 2008, Proceedings*. IEEE-CS Press, August 2008.

A Redundant S-Box lookup

A.1 Mapping the coefficients to $\mathbb{F}_D$

Let the coefficients of the idempotent i_1 be i_{11} and i_{12}. Analogously, $i_2 = i_{21} \cdot y + i_{22}$. Then every ring element $u \in \mathbb{U}$ can be written as

$$u = (\phi_1(x_D)i_{11} + \phi_2(x_C)i_{21}) \cdot y + (\phi_1(x_D)i_{12} + \phi_2(x_C)i_{22}).$$

If x_C is known, then $\phi_2(x_C)i_{21} \cdot y + \phi_2(x_C)i_{22}$ can be subtracted from u. What remains is the polynomial

$$\phi_1(x_D)i_{11} \cdot y + \phi_1(x_D)i_{12}$$

which contains x_D in both coefficients. Furthermore, as long as i_{11} and i_{12} are co-prime to P, there exists a one-to-one mapping between the coefficients and x_D.

A.2 Redundant S-Box lookup in detail

For reasons of simplicity we omit the mappings ϕ_1 and ϕ_2 here. The input of the algorithm is the polynomial

$$\begin{aligned} \mathrm{CRT}(x_D, x_C) = \\ u = u_1 y + u_0 = \\ (x_D i_{11} + x_C i_{21}) \cdot y + (x_D i_{12} + x_C i_{22}) \end{aligned}$$

In line 1 the signature x_C is extracted:

$$\begin{aligned} t = u \cdot i_2 = \\ x_D i_1 i_2 + x_C i_2 i_2 \pmod{p_1 \cdot p_2} = \\ x_C i_2 \end{aligned}$$

Afterwards, the difference $d = (x_{C_{in}} + x_C) \cdot i_2$ is calculated in line 2. This difference is then used to normalize the input signature of u to $x_{C_{in}}$ in line 3. The normalized u is denoted as u'.

The redundant S-Box itself takes $x_D i_{11} + x_{C_{in}} i_{21}$ and outputs

$$\begin{aligned} (\mathrm{SubBytes}(x_D)i_{11} + x_{C_{out}} i_{21}) \cdot y + \\ \mathrm{SubBytes}(x_D)i_{12} + x_{C_{out}} i_{22} \quad + \\ \mathrm{error}(x_D) \end{aligned}$$

The correction table CT takes $x_D i_{12} + x_{C_{in}} i_{22}$ and outputs $\mathrm{error}(x_D)$. After adding up the output of the redundant S-Box SB, the output of the correction table CT, and the difference d, the algorithm yields

$$\begin{aligned}(\mathrm{SubBytes}(x_D) i_{11} + (x_{C_{out}} + x_{C_{in}} + x_C) i_{21}) \cdot\, y + \\ \mathrm{SubBytes}(x_D) i_{12} + (x_{C_{out}} + x_{C_{in}} + x_C) i_{22} \quad = \\ \mathrm{CRT}(\mathrm{SubBytes}(x_D), x_C + x_{C_i} + x_{C_o})\end{aligned}$$