Protecting AES Software Implementations on 32-Bit Processors Against Power Analysis

Stefan Tillich, Christoph Herbst, and Stefan Mangard

Graz University of Technology,
Institute for Applied Information Processing and Communications,
Inffeldgasse 16a, A–8010 Graz, Austria
{Stefan.Tillich,Christoph.Herbst,Stefan.Mangard}@iaik.tugraz.at

Abstract. The Advanced Encryption Standard is used in many embedded devices to provide security. In the last years, several researchers have proposed to enhance general-purpose processors with custom instructions to increase the efficiency of cryptographic algorithms. In this work we have evaluated the impact of such instruction set extensions on the implementation security of AES. We have compared several AES implementation options which incorporate state-of-the-art software countermeasures against power-analysis attacks—with and without the use of instruction set extensions. For both scenarios we provide a thorough analysis for different countermeasures with regard to security, performance, and memory. We have found that even a moderate level of protection requires a considerable overhead both in terms of speed and memory. The instruction set extensions, which have been solely designed to increase performance, help to reduce this overhead, but it still remains high. An implementation with proper protection through software countermeasures is only feasible in a setting where the need for resistance against power analysis outweighs the need for performance.

Keywords: Advanced Encryption Standard, side-channel attacks, software countermeasures, instruction set extensions, implementation security, DPA, power analysis.

1 Introduction

Today, more and more computational tasks are performed on small embedded systems. Most of these systems feature an embedded processor with a wordsize of 8, 16, or 32 bit. 32-bit processors are common in mid-range to high-end embedded systems like PDAs and cellphones but can also be found in wireless sensor networks and even in some high-end smartcards. Many applications require the execution of cryptographic algorithms in order to achieve some security assurances, e.g. data confidentiality or authentication. But while the algorithms themselves are secure, a straightforward implementation on a device is very likely to be vulnerable to side-channel attacks. Such attacks measure and analyze some physical property of the device while it performs cryptographic operations, with the goal of extracting the key used by the device. Power-analysis attacks, which exploit the power consumption, have been studied very thoroughly, and many proposals have been made on how to make them more effective as well as on how to defend against them.

J. Katz and M. Yung (Eds.): ACNS 2007, LNCS 4521, pp. 141–157, 2007.

While it is unlikely that side-channel attacks can be fully prevented, appropriate countermeasures can hamper an attack to the point where it becomes practically infeasible. Some hardware countermeasures have proven to be rather effective in doing this. On the other hand, counteracting power-analysis in software is very hard, as the programmer normally has only a very limited influence on the power consumption of the processor. To make things even worse, in the last years new attack variants have emerged, which are very effective against software implementations of cryptographic algorithms.

We have investigated the current situation regarding software countermeasures against state-of-the-art power-analysis attacks. We have focused on 32-bit embedded processors and on the AES algorithm, but most of the discussed methods also work for processors of different wordsize and other cryptographic algorithms. This paper is organized as follows: In Section 2 we discuss software countermeasures for power analysis in principal. We elaborate on the effectiveness of these countermeasures in Section 3. Section 4 focuses on state-of-the-art attacks on protected software implementations. In Section 5 we estimate the effect of different countermeasures on performance, memory, and implementation security. We draw conclusions in Section 6.

2 Power-Analysis Countermeasures for Software Implementations

In order to secure software implementations of cryptographic algorithms against power-analysis attacks there are two suitable approaches, namely masking and hiding. Masking conceals all intermediate values during the calculation with a random mask. Hiding techniques try to break or at least weaken the link between processed intermediate values and the side-channel leakage at a certain moment of time. In this section we give an overview of these two types of countermeasures.

2.1 Masking

Masking means to conceal each intermediate value a with a random value m, which is called mask. These masks are generated by the device for each execution of the algorithm and they are not known by an attacker. Generally, we can distinguish between Boolean and arithmetic masking. In arithmetic masking, intermediate values and masks are combined with an arithmetic operation like addition or multiplication. For AES, Akkar et al. suggested a multiplicative masking scheme [1] where the intermediate values are concealed with a multiplicative mask $a_m = a \cdot m \pmod{n}$. Boolean masking uses the exclusive-or operation to combine intermediate values and masks $a_m = a \oplus m$. Masking schemes for software implementations of AES based on Boolean masking have been proposed in [5] and [6].

Power-analysis attacks are prevented by masking because each intermediate value is masked with a random value and thus the power consumption caused by this value can not be predicted by an attacker. This holds under the condition, that each masked value a_m is independent of a. Usually the masks are applied to the plaintext values at the beginning of the algorithm. During the execution of the algorithm one has to keep track of the modification of the masks by the operations of the algorithm. For

the AES operations ShiftRows and AddRoundKey, this can be done with virtually no effort, because they are linear and do not change the applied masks. The MixColumns operation combines different values of one column of the AES State, and therefore, one has to calculate the modified masks after this operation. To monitor the change of the masks through the nonlinear SubBytes transformation, a more elaborated approach is needed. A very common way to implement the SubBytes transformation in software is to use lookup tables: $a_{out} = S(a_{in})$, where S denotes the AES S-box, which is used on every byte of the State for SubBytes. In order to mask the SubBytes transformation, we have to calculate a masked table S' such that $S'(a_m) = S'(a \oplus m) = S(a) \oplus m'$. When implementing such a masking scheme, care has to be taken that all intermediate values stay masked during the critical computations of the algorithm. At the end of the calculation of the algorithm all masks have to be removed. Especially unintended unmasking has to be considered. This can happen in a device which leaks the Hamming distance of subsequently processed values, i.e. the Hamming weight of the exclusive-or of these two values [12]. Two subsequent values with the same Boolean mask would therefore be unmasked.

Provably secure masking schemes for AES have been published in [2] and [15]. These schemes focus on hardware implementations. In [16] a proposal for a software implementation of the scheme presented in [15] has been made. This scheme has higher performance rates than a conventional lookup scheme, as long as a set of masks is only used for a single encryption. In schemes where masks are used for more than one encryption, the lookup table approach is still faster. This is one of the reasons why we have chosen a lookup based scheme for our implementation.

Masking schemes are an appropriate choice to defeat first-order power-analysis attacks. Nevertheless, masked implementations are still vulnerable to higher-order and template attacks. Higher-order attacks are discussed in [14], [19], [17], and [8]. A template based attack on a protected AES software implementation has been published by Oswald et al. in [13]. Due to the presence of these powerful attacks it is mandatory to combine masking schemes with a second type of countermeasures to raise the level of security.

2.2 Hiding

In general, hiding can take place in two domains, namely in the time domain and in the amplitude domain. Hiding in the time domain tries to randomize the time of occurrence of a specific operation, whereas hiding in the amplitude domain tries to reduce the effect of the performed operation on the overall power consumption.

For software implementations, hiding in the time domain is normally easier to achieve. The goal is to distribute the occurrence of critical operations and intermediate values over a given period during each execution of an algorithm. This leads to a reduced correlation of targeted values at specific points of time. Two appropriate methods to achieve this randomization are the insertion of dummy operations and shuffling of operations. Both insertion and shuffling are controlled by random values generated by the device. Inserted dummy operations should not be distinguishable from normal operations. Otherwise an attacker could be able to remove their effect from the power trace. Shuffling of operations means that for each execution of the algorithm, the order

of the occurring intermediate values is changed. How these two methods can be applied to a software implementation of AES is described in Section 3.3.

Hiding in the amplitude domain is rather hard for software implementations. Nevertheless, a designer has the opportunity to choose only such instructions which leak a minimum amount of information. This technique highly depends on the used device and its leakage properties. The statistic effects of hiding have been investigated in [11], [4], and [3].

3 Effectiveness of Software Countermeasures

This section gives a thorough evaluation of software countermeasures that can be applied to secure an AES implementation on a 32-bit platform. In this context we have considered two classes of processing platforms. The first class consists of typical 32-bit embedded processors with a standard RISC instruction set architecture. The second class includes processors which have explicit support for cryptographic operations in their instruction set (cryptographic instruction set extensions).

For the instruction set extensions we have used the "advanced word-oriented AES extensions with implicit ShiftRows" described in [18]. These instructions work on 32-bit words performing either four parallel AES S-box lookups (`sbox4s/isbox4s/sbox4r`) or a MixColumns transformation for a single State column (`mixcol4s/imixcol4s`). In the following, we will give the definition of the functionality of the `sbox4s` and `mixcol4s` instructions. Note that *rs1* and *rs2* denote the two 32-bit input operands and *rd* the 32-bit result of the instruction. Brackets with indices are used to select a part of the respective 32-bit value, while | concatenates four 8-bit or two 16-bit values to a 32-bit value. *S-box* substitutes an 8-bit value according to the AES S-box, while *MixColumns* transforms a 32-bit value following the AES MixColumns operation.

sbox4s rs1, rs2, rd: rd[31..0] := S-box(rs1[31..24]) | S-box(rs2[23..16]) | S-box(rs1[15..8]) | S-box(rs2[7..0]);
mixcol4s rs1, rs2, rd: rd[31..0] := MixColumns(rs1[31..16] | rs2[15..0]);

The definition of `isbox4s` and `imixcol4s` is similar, with the difference that the inverse AES S-box and the InvMixColumns transformation are used, respectively. Finally, the `sbox4r` instruction has only one input operand, whose bytes are transformed with the AES S-box and where the result is rotated 8 bits to the left:

sbox4r rs1, rd: rd[31..0] := S-box(rs1[23..16]) | S-box(rs1[15..8]) | S-box(rs1[7..0]) | S-box(rs1[31..24]);

The `sbox4r` instruction is designed for use in the AES key schedule, while the other instructions are intended to speed up the AES round transformations.

In the following sections we analyze different options for power-analysis countermeasures. The most powerful attacks are listed and implementation-specific details for use of the instruction set extensions are given. The maximum correlation coefficient ρ is stated for each attack. The security gain can be approximated by the quotient of the correlation coefficient for an attack on an unprotected and on a protected implementation: An attack on the protected implementation requires at least $(\frac{\rho_{unprotected}}{\rho_{protected}})^2$ more

power traces [12]. For our estimations we have $\rho_{unprotected} = 1$ and can therefore state the security gain as $(\frac{1}{\rho})^2$, where ρ always denotes the correlation coefficient for an attack on the protected implementation. Note that we state ρ for noise-free environments, which is sufficient to make a relative comparison of unprotected and protected implementations. The correlation coefficients observed in practical attacks will be lower due to noise. The correlation coefficient has been determined under the assumption that the Hamming weight of processed values leaks through the power consumption. Many devices leak the Hamming distance of subsequently processed values, but it is very hard to determine the correlation coefficient for such a setting without taking many details of the processor architecture and software implementation into account. We therefore take the Hamming-weight leakage model as a lower bound for devices that leak the Hamming distance. This assumption holds as long as the software implementation avoids potential vulnerabilities due to the Hamming-distance leakage, e.g. unintended unmasking as explained in Section 2.1.

3.1 Unprotected Implementation

An unprotected 32-bit AES software implementation is vulnerable to a multitude of attacks. One of the most powerful attacks is a first-order DPA on an 8-bit intermediate result after the S-box lookup ($\rho = 1$). The key expansion can also be targeted directly with a template-like attack as described in [10]. This attack extracts the Hamming weights of 8-bit intermediate values of the key expansion and uses the dependency of these values to narrow down the number of potential keys. The use of the instructions set extensions from [18] allows to calculate the key schedule with 32-bit values only, which makes this kind of attack infeasible.

3.2 Masking

A masked implementation protects critical intermediate values with a random mask. An intermediate value of the AES operation can be considered critical when it depends on a small portion of the (round) key and on the plaintext or ciphertext. In this case the attacker can guess the part of the key and verify her guess through analysis of the measured power traces. The choice of masks and the processing order of masked values must always be done carefully with regard to the leakage of the device to prevent problems like unintended unmasking.

If the masking countermeasure is implemented properly, it can prevent first-order DPA attacks. However, a masked implementation is still vulnerable to higher-order DPA attacks. In such an attack, several points of each power trace are combined to a single value with a preprocessing function *pre*. The resulting value again depends on some predictable value. The preprocessed values can then be subjected to a first-order DPA attack. Normally, a second-order DPA attack is sufficient to break a masked implementation. The targeted values for preprocessing are either a masked intermediate value and the corresponding mask, or two intermediate values with the same mask.

The best vantage point to break a masked AES implementation is the masked S-box lookup, which is used for SubBytes. This lookup requires masked 8-bit input and output values, which are easier to target than the masked 32-bit values resulting from other

transformations (e.g. MixColumns). The cost for precomputing a single masked S-box is very high, and it is therefore necessary to reuse masked S-box tables. This results in the processing of 8-bit values with the same mask, which can be targeted in a second-order attack ($\rho = 0.24$, see [12]). But even if no 8-bit value carries the same mask, the preprocessing function could use the power consumption of the mask itself (which must occur at some time in the computation) as second value. In the worst case for the attacker, this 8-bit mask will only occur in form of a 32-bit word, where the other 24 bits are random (this can only be achieved with the help of the instruction set extensions). Even in this case, the level of protection against a second-order DPA attack is rather low ($\rho \approx 0.1$).

A possibility to prevent the S-box lookup in software is to perform most of the AES round as table lookup (T-box lookup). However, the precomputation of masked T-boxes would be much more costly than the precomputation of masked S-boxes. Moreover, a T-box lookup still requires an 8-bit masked input value, which can be targeted in an attack.

A masked implementation could use more than one mask for every intermediate value. However, it seems very difficult to generate a masked S-box table without processing the definite input and output masks at some point of time. All in all, the vulnerability to second-order DPA attacks is very hard to remove in a masked AES implementation.

3.3 Randomization

In the following, we will denote countermeasures of hiding in the time domain (cf. Section 2.2) as *randomization*. In a randomized AES implementation, the occurrence of a specific intermediate value at a specific point in time is reduced to a certain probability. This can be done by shuffling of operations and by random insertion of dummy operations. In this case an attacker needs to capture more power traces, in order to compensate for this uncertainty.

Simple solutions, like the random insertion of `nop` instructions, are likely to be detected and removed by an attacker. Therefore, if dummy operations are added, it is important that they can not be distinguished from the genuine operations. This can be achieved by performing the AES transformations on some dummy data.

The best degree of randomization can be achieved by using both the shuffling of operations and the insertion of dummy operations. In AES, the smallest unit of data, whose processing can be randomized, is the 8-bit input and output value used in the S-box lookup. The 16 S-box lookups per AES round can therefore be shuffled, resulting in a probability of $p = \frac{1}{16}$ for a specific value at a specific point in time. Dummy operations can be inserted by processing a certain number of dummy values. Processing of complete dummy States (i.e. 4×4-byte matrices) seems to be a good granularity for that purpose. If N dummy States are processed in addition to the genuine State, then the probability for the occurrence of a specific value goes down to $p = \frac{1}{(N+1)\cdot 16}$.

It would be very inefficient to perform a selection for each of the $(N+1)\cdot 16$ byte values separately. Moreover, the AES algorithm does not allow to perform all critical round transformations with just a single byte. The smallest value which is sufficient for all those transformations is a single State column. For practical implementation

it is sufficient to determine the processing order of the bytes in an orthogonal way: The States are processed one after the other, i.e. the processing of the genuine State is randomly embedded within the processing of the dummy States. For each State, the columns are processed in a fixed order beginning with a randomly chosen column. For each column, the bytes are processed separately and also in a fixed order starting with a randomly chosen byte.

The randomization degree p determines the resistance against DPA attacks. The power traces obtained from an implementation with randomization are often referred to misaligned power traces. A direct DPA attack on the misaligned traces would require $(\frac{1}{p})^2$ more traces to compensate for the randomization. However, Clavier et al. [4] have proposed to sum up all points in the power trace, where the targeted value can occur. This approach is often referred to as *"windowing"*. With this approach, an attacker only requires $\frac{1}{p} = (N+1) \cdot 16$ more traces to defeat the randomization.

Therefore we can assume that the number of power traces to attack a randomized AES implementation scales up with a factor of only $(N+1) \cdot 16$, as $\rho = \frac{1}{p} = \frac{1}{(N+1) \cdot 16}$. Most of the overhead of a randomized implementation comes from the preparation of the randomization and the byte-wise processing of the AES State. Doubling the security (which corresponds to doubling of $(N+1)$) roughly doubles the total running time. This results in a very large overhead.

3.4 Masking and Randomization

For better protection, an AES implementation needs to combine masking and randomization countermeasures. However, there are still several possible attacks which can break such an implementation rather efficiently.

An attacker will try to defeat the masking with a second-order attack. At least one of the attacked intermediate values (i.e. a masked 8-bit value) is protected by randomization. As we have already outlined in Section 3.3, a very effective way to defeat randomization is to sum up the power consumption at all moments in time where the attacked value can occur (recall that for our considered randomization there are $(N+1) \cdot 16$ points in time, where N is the number of dummy States). The second attacked value can either be the mask of the first value, or another randomized intermediate value carrying the same mask as the first value.

There are two main strategies on how to use the second value in an attack. On the one hand, this value can be employed to introduce a bias in the occurring masks. On the other hand, the value can be combined with the first one to yield a result that depends on the unmasked value.

4 Attacks on Masked and Randomized AES Implementations

We have shown in the previous section, that a protection by masking or randomization alone can not withstand power-analysis attacks. In this section we analyze the possible attacks on software implementations which use a combination of both countermeasures. The attacks presented in this section have either been published and evaluated or are natural extensions or combinations of existing attacks. The method of windowing

(cf. Section 3.3) published by Clavier et al. [4] is fundamental for all of the examined attacks, as it is a very good way to compensate the effects of the randomization countermeasure. The possibility of second-order DPA attacks has already been mentioned in the original publication of Kocher et al. [9]. Second-order attacks on software implementations of block ciphers have been analyzed in [14].

In [13], Oswald et al. have evaluated the effectiveness of template-based attacks against masked software implementations and have shown that such methods can be very effective. However, as long as the targeted operation used for template-building remains randomized in time, we assume that it is very hard to create well-matching templates, which lead to better results than techniques based on counteracting randomization, e.g. windowing.

4.1 Biasing Masks

A very powerful attack is to introduce a bias into the masks used by the device, which leads to a dramatic decrease of security. This idea has been introduced by Jaffe [7], and practically evaluated by Oswald et al. [13]. In practice, an attacker can bias a mask by examining a point of the power trace where the mask is processed and by discarding all traces which have a value above (or below) a certain threshold. Figure 1 shows the timeline of a power trace, where the time of occurrence of targeted values is marked at the top. Below the timeline, it is shown how the power consumption values at these times would be used in a biased-mask attack. Windowing is used to sum up the power consumption at all points in time in the selected traces where the attacked value can occur (due to randomization). A classical first-order DPA attack is performed on the resulting preprocessed power values.

Without instruction set extensions, the 8-bit masks of the S-box can be targeted directly during the generation of the masked S-box. With instruction set extensions, the masked S-box can be generated using only 32-bit masks (provided that four masked S-boxes are used). A bias of either the 8-bit or 32-bit mask has a devastating effect on the security. For example, biasing the 8-bit masks to a Hamming weight (HW) ≥ 6 yields $\rho = -0.1$ (for $N = 1$). For 32-bit masks, a bias of HW ≥ 20 results in $\rho = -0.05$ (again for $N = 1$).

Increasing the degree of randomization does not lower the correlation coefficient very effectively (see Table 2). Note that a possible defence against this attack could consist of randomizing the time of occurrence of each mask. However, the mask and values directly dependent on the mask occur at several points in the computation, e.g. generation of the mask, appliance of the mask to the S-box, calculation of the mask after MixColumns, (re)masking of the key schedule. Proper randomization of all these operations would be quite challenging and also incur a considerable overhead in terms of performance.

4.2 Combining Second-Order DPA and Windowing

A second-order DPA can be combined with windowing to break the masking and randomization. This approach can be seen as performing multiple second-order DPA attacks in parallel. The attack can be done by combining the power consumption for the

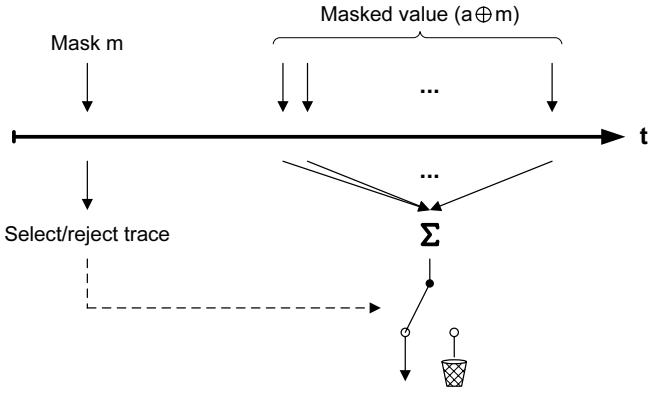

Fig. 1. Information extraction from power traces in a biased-mask attack

mask processing with each of the $(N+1)\cdot 16$ points in time where the targeted masked value can occur (due to randomization) using a second-order preprocessing function.

However, due to the randomization, the attacker does not know which of the resulting values corresponds to her targeted value. This is the same problem as in an implementation which has only randomization countermeasures. Consequently, an efficient solution is to sum up all $(N+1)\cdot 16$ preprocessed values and to perform a first-order DPA attack on the result. Figure 2 depicts this approach.

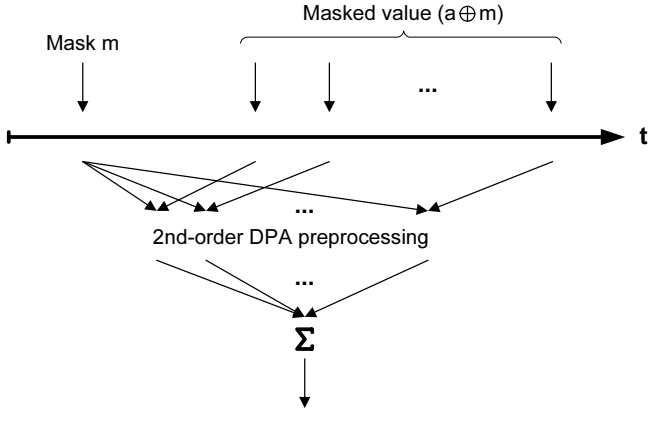

Fig. 2. Information extraction from power traces for second-order DPA combined with windowing

The effectiveness of this attack can be evaluated for both attack stages separately. In the first stage, the second-order DPA preprocessing function is applied to each pair of values (mask and masked value). For our randomization scheme we have an 8-bit

masked value. As already stated in Section 3.2 we have $\rho = 0.24$ for 8-bit masks and $\rho \approx 0.1$ for 32-bit masks (using instruction set extensions). The summation of the second attack stage corresponds to windowing, which scales down the correlation coefficient with a factor of $\frac{1}{\sqrt{(N+1)\cdot 16}}$. The overall correlation coefficient is therefore very high: For the 8-bit masks we get $\rho = \frac{0.24}{\sqrt{(N+1)\cdot 16}}$, and for 32-bit masks we get $\rho \approx \frac{0.1}{\sqrt{(N+1)\cdot 16}}$. So in order to achieve $\rho = 0.01$, we would need at least $N = 5$.

Principally, it would be desirable to randomize the occurrence of the mask to the same degree as the masked value. This measure would require to sum up all possible combinations where mask and masked value can appear. The number of combinations is $((N+1)\cdot 16)^2$, which would lead to a reduction of the correlation by a factor of $(N+1)\cdot 16$ after windowing. At $N = 1$, the correlation would already be about as low as $\rho = 0.003$ for 32-bit masks. However, as already mentioned in Section 4.1, randomization of the mask would be very costly in terms of performance.

4.3 Targeting Weak Randomization

Targeting two randomized intermediate values which carry the same mask is normally less efficient than to target one fixed (e.g. the mask) and one randomized value. However, a weak randomization can be broken more easily with this strategy.

In this context, a weak randomization is one where two intermediate values with the same mask always occur with a fixed distance in time. An example for this are the S-box inputs of the first and second AES round, when the used S-boxes have the same input masks and the two lookups are not randomized separately. The attacker can therefore apply the second-order DPA preprocessing function to each such pair of values, which is depicted in Figure 3. The rest of the attack is exactly the same as the previously described one (summation followed by first-order DPA).

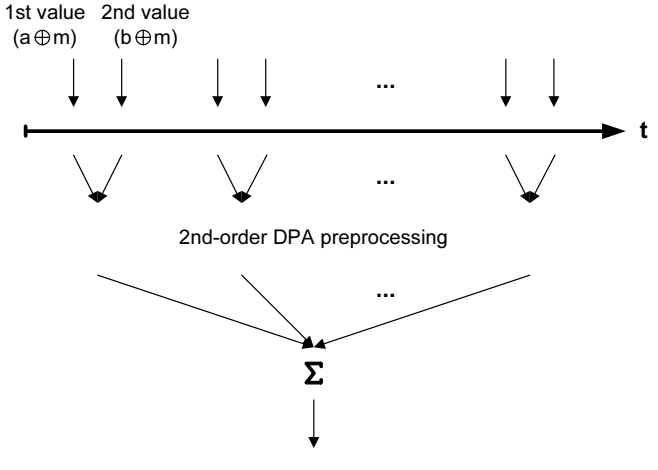

Fig. 3. Information extraction from power traces when attacking a weak randomization

Targeting two 8-bit intermediate values with the same mask is equivalent to targeting one intermediate value and the according 8-bit mask. The correlation coefficient is therefore $\rho = \frac{0.24}{\sqrt{(N+1)\cdot 16}}$. However, it might be necessary to hold some parts of the plaintexts constant and guess more than a single key byte to be able to set up a good hypothesis.

The effectiveness of this attack can again be evaluated for both attack stages independently. In the first stage, the second-order DPA preprocessing function is applied to each pair of values with the same mask. For our randomization scheme we have two masked 8-bit values, which yields $\rho = 0.24$ [12]. The summation of the second attack stage again corresponds to windowing, which reduces the correlation coefficient by a factor of $\frac{1}{\sqrt{(N+1)\cdot 16}}$. The resulting correlation coefficient remains rather high with $\frac{0.24}{\sqrt{(N+1)\cdot 16}}$ (e.g. $\rho = 0.01$ would require $N = 35$).

To counteract this attack it would be necessary to randomize the S-box lookup in the first and second AES round separately (and similarly in the two last rounds). This countermeasure would render the described attack less efficient than the other described attacks.

4.4 "Classical" Second-Order DPA on Windowed Traces

Another way to combine second-order DPA and windowing is to perform windowing first to counteract the effects of randomization, and to do a "classical" second-order DPA attack on the result. Figure 4 depicts the processing steps performed on every power trace. The resulting value can then be subjected to a first-order DPA attack. A preprocessing function *pre*, which is generally very effective, is the absolute difference of the inputs: $pre(a,b) = |a-b|$ [12]. For this function, it is important that both a and b are of the same magnitude, e.g. if a is a single point from the trace and b is a sum of n points, then the preprocessing function should scale a up to b: $pre(a,b) = |n \cdot a - b|$.

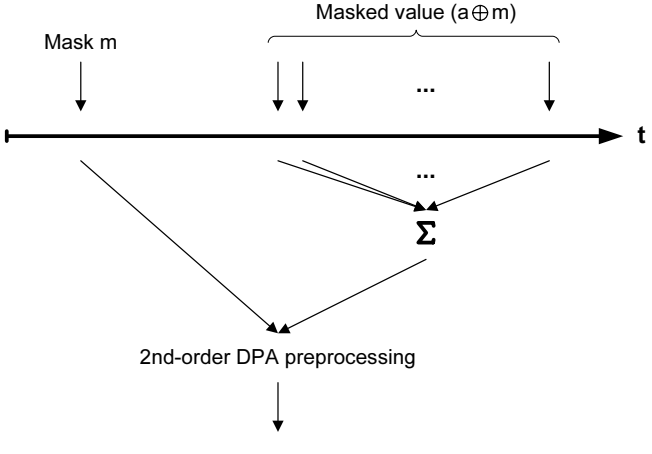

Fig. 4. Information extraction from power traces in a "classical" second-order DPA

For a randomization degree of $N = 1$, the correlation is about $\rho = 0.013$ for 8-bit masks and $\rho = 0.012$ for 32-bit masks. Doubling the randomization degree approximately halves the correlation coefficient.

5 Performance Estimation

The security evaluation of the last section has shown that there are powerful attacks which can break implementations even when they employ very sophisticated countermeasures. Under the assumptions of our analysis, one might be inclined to regard the use of software countermeasures as futile. Nevertheless there are scenarios, where a protected implementation might be desired, even if the provided protection is rather moderate:

- In a device with a fixed processor, the use of software countermeasures is likely to be the only available option. In some applications, a certain degree of implementation security could still be much better than none at all.
- The most powerful attacks used in our security evaluation might not be applicable due to other security measures of the device (e.g. limited number of AES encryption/decryptions, plaintext/ciphertext not selectable by the attacker, etc.).
- The device has some hardware countermeasures (e.g. noise generators) and the resistance against power-analysis should be amplified by the software countermeasures.

In order to provide performance estimations for different countermeasures, we have implemented AES-128 encryption with both masking and a scalable randomization. With the help of this implementation we have estimated the performance for several design options and degrees of randomization. First, we present the most important design decisions and implementation characteristics of our solution. Then we give the performance figures for interesting implementation variants regarding expected security level, speed, and memory requirements.

5.1 Features of Our Protected AES Implementation

Some basic design decisions for our 32-bit implementation are similar to the secure AES implementation for 8-bit microcontrollers presented in [5]. This mainly concerns the basic types of countermeasures (masking and randomization), the concept of randomized zones, etc. We assume the availability of a random number generator to provide mask values and randomization parameters.

The masking scheme requires six distinct byte masks as input. Two mask bytes are used to derive a masked S-box lookup table with input mask M and output mask M'. The four other bytes (denoted $M1$, $M2$, $M3$, and $M4$) mask each input column to the MixColumns transformation. The corresponding output masks can be derived by performing MixColumns on the mask values alone. More precisely, $M1$ to $M4$ are used as an input column for the MixColumns transformation, resulting in the output masks $M1'$, $M2'$, $M3'$, and $M4'$.

All operations which yield intermediate results depending on a relatively small portion of the key are executed in a randomized fashion. Randomization is achieved both by shuffling of operations as well as the addition of dummy operations. The processing of the AES State is shuffled so that each byte is processed at one of 16 moments in time with equal probability. Dummy operations are inserted as normal AES round tranformations, but work on a random State (*dummy State*). The processing of the *genuine State* is randomly embedded in between the processing of several dummy States. The parts of the encryption where execution is randomized are denoted as *randomized zones*. The randomized zone at the beginning of AES encryption reaches up to and including the SubBytes operation of round 2, while the randomized zone at the end starts with SubBytes in round 9. Figure 5 gives a general overview of the program flow for the AES implementation and shows the masks on the State as well as the randomized transformations.

Randomization of operations is costly in terms of performance. Therefore it is desirable to keep the randomized zones as short as possible. In our implementation we

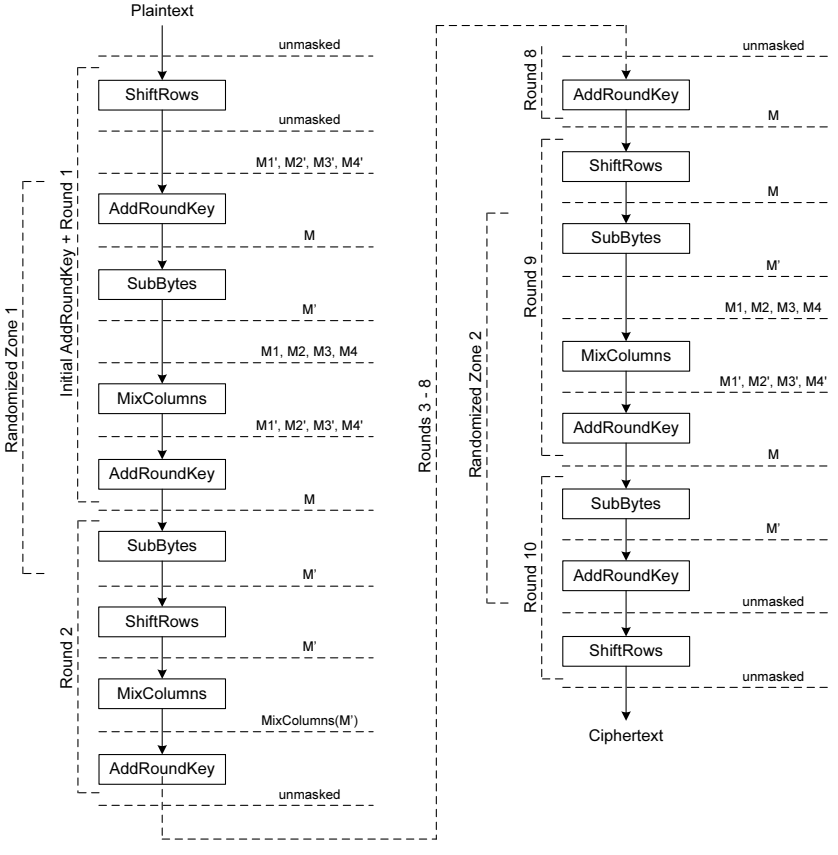

Fig. 5. Program flow of masked and randomized AES encryption

Table 1. Performance and RAM requirements of AES-128 encryption implementations

Countermeasures	Pure Software	ISE	Memory (RAM)
	cycles	cycles	bytes
None	1,637	196	176
1SB_WR	$6,465 + 1,888N$	$2,023 + 1,028N$	476
4SB_WR	$14,958 + 1,888N$	$3,631 + 1,028N$	1248
4SB_SR	$15,332 + 2,208N$	$3,978 + 1,348N$	1388

have reordered the round transformations, so that ShiftRows is not included in the randomization. This reordering requires the first and last round key to be transformed with ShiftRows or InvShiftRows.

In order to reduce the overhead for masking, the AddRoundKey operation is used for remasking whenever possible. This requires masks to be applied on some of the round keys. The masks on these round keys must be renewed whenever the masks change. When the masks are changed for each AES encryption—which is the ideal case—then it would be equally efficient to change the mask explicitly during the AES encryption.

In our implementation, the rounds 3 to 8 are not masked. AddRoundKey of round 2 removes the masks from the State, and AddRoundKey of round 8 masks the State again. All unmasked intermediate values have therefore been subjected to three AddRoundKey transformations and depend on sufficiently many key bytes, to prevent an efficient DPA attack. The advantage of the unmasked inner rounds is that the AES instruction set extensions can be fully used.

The randomization follows the concepts described in Section 3.3. In the randomized zones, only values which depend on a single State byte are processed. This allows for a randomization degree of $(N+1) \cdot 16$, where N is the number of dummy States.

5.2 Performance Figures

Table 1 contains the execution times and RAM requirements for several implementations of AES-128 encryption with masking and randomization countermeasures. The performance figures are given for the case without instruction set extensions (pure software) as well as with instruction set extensions (ISE). The RAM requirements for a specific implementation is always the same for both cases. The cycle counts are given in dependence on the number of dummy States (N).

We have given performance figures for three protected implementations, which employ both masking and randomization countermeasures. The cycle counts include all overhead when the masks are refreshed for each new encryption. The first implementation (1SB_WR) uses 1 masked S-box and a weak randomization (weak in the sense defined in Section 3.4). The second implementation (4SB_WR) is similar, but uses 4 masked S-boxes. The last implementation (4SB_SR) has a strong randomization.

For comparison, the performance figures of an unprotected implementation, as stated in [18], are provided.

Table 2 gives a complete analysis of the security/performance trade-off for the three protected implementations. Note that SW denotes the software implementation, while ISE denotes the respective implementation with instruction set extensions. The table

Table 2. Analysis of the security/performance trade-off

Implementation	Performance	BM	2W	WR	W2	max(ρ)
1SB_WR (SW), $N = 0$	6,465	-0.14	0.06	0.06	0.03	-0.14
1SB_WR (SW), $N = 3$	12,129	-0.07	0.03	0.03	< 0.01	-0.07
1SB_WR (SW), $N = 5$	15,905	-0.06	0.02	0.02	< 0.01	-0.06
1SB_WR (SW), $N = 11$	27,233	-0.04	0.02	0.02	< 0.01	-0.04
1SB_WR (ISE), $N = 0$	2,023	-0.14	0.06	0.06	0.03	-0.14
4SB_WR (ISE), $N = 0$	3,631	-0.05	0.03	0.06	0.02	0.06
4SB_SR (ISE), $N = 0$	3,978	-0.05	0.03	N/A	0.02	-0.05
4SB_SR (ISE), $N = 1$	5,326	-0.04	0.02	N/A	0.01	-0.04
4SB_SR (ISE), $N = 3$	8,022	-0.03	0.01	N/A	< 0.01	-0.03
4SB_SR (ISE), $N = 5$	10,718	-0.02	0.01	N/A	< 0.01	-0.02
4SB_SR (ISE), $N = 11$	18,806	-0.01	< 0.01	N/A	< 0.01	-0.01

lists the estimated correlation coefficients for the four attacks presented in Section 3.4: Biased mask attack (BM), combined second-order DPA and windowing attack (2W), weak randomization attack (WR), and "classical" second-order DPA attack on windowed traces (W2). The maximum correlation coefficient is listed in the last column.

For the pure software implementation, the biased-mask attack (BM) is the most powerful one. In software, the only option is to increase the randomization degree N. But the correlation coefficient only decreases very slowly with rising N. When instruction set extensions are available, we can work exclusively with 32-bit masks if we use four masked S-boxes instead of one (4SB_WR). In that case, the attack exploiting the weak randomization becomes the most efficient one. To counteract, we use the implementation with strong randomization (4SB_SR), which makes this attack inapplicable. Then the biased-mask attack becomes again the most effective one. With heavy randomization ($N = 11$), the correlation coefficient can be pushed down to $\rho = -0.01$. This corresponds to an increase of the security level by four orders of magnitude in comparison to an unprotected implementation. This comes at the price of an execution time, which is increased by two orders of magnitude (cf. Table 1). Compared to the unprotected pure software implementation, the execution time is increased by one order of magnitude.

6 Conclusions

In this paper we have provided a thorough analysis of power analysis countermeasures in software in the face of state-of-the-art attacks. We have concentrated on 32-bit embedded processors, but most of the results could also be applied to 8-bit and 16-bit processors. By means of an AES implementation we have shown the impact of power analysis countermeasures on the performance and RAM requirements. When restricted to the original instruction set architecture, the attainable degree of protection of our protected implementation is increased by three orders of magnitude. If the processor is equipped with custom instructions for AES, then a protection level of four orders of magnitude is achievable. But the performance penalty is rather high, so that it is probably not acceptable for all applications. As of now, no set of software countermeasures seems suited to offer a reasonable degree of protection at a negligible overhead.

Future Work. The use of existing instruction set extensions for AES is not sufficient to support power analysis countermeasures. A promising approach which we will investigate in the future is to enhance the extensions with hardware countermeasures.

Acknowledgements. The research described in this paper has been supported by the Austrian Science Fund (FWF) under grant number P18321-N15 ("Investigation of Side-Channel Attacks") and P16952-N04 ("Instruction Set Extensions for Public-Key Cryptography"), by the European Commission under grant number FP6-IST-033563 (Project SMEPP) and, in part, by the European Commission through the IST Programme under contract IST-2002-507932 ECRYPT. The information in this document reflects only the authors' views, is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

References

1. M.-L. Akkar and C. Giraud. An Implementation of DES and AES, Secure against Some Attacks. In Çetin Kaya Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 309–318. Springer, 2001.
2. J. Blömer, J. Guajardo, and V. Krummel. Provably Secure Masking of AES. In H. Handschuh and M. A. Hasan, editors, *Selected Areas in Cryptography, 11th International Workshop, SAC 2004, Waterloo, Canada, August 9-10, 2004, Revised Selected Papers*, volume 3357 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2005.
3. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In M. J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.
4. C. Clavier, J.-S. Coron, and N. Dabbous. Differential Power Analysis in the Presence of Hardware Countermeasures. In Çetin Kaya Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, volume 1965 of *Lecture Notes in Computer Science*, pages 252–263. Springer, 2000.
5. C. Herbst, E. Oswald, and S. Mangard. An AES Smart Card Implementation Resistant to Power Analysis Attacks. In J. Zhou, M. Yung, and F. Bao, editors, *Applied Cryptography and Network Security, Second International Conference, ACNS 2006*, volume 3989 of *Lecture Notes in Computer Science*, pages 239–252. Springer, 2006.
6. K. Itoh, M. Takenaka, and N. Torii. DPA Countermeasure Based on the Masking Method. In K. Kim, editor, *Information Security and Cryptology - ICISC 2001, 4th International Conference Seoul, Korea, December 6-7, 2001, Proceedings*, volume 2288 of *Lecture Notes in Computer Science*, pages 440–456. Springer, 2002.
7. J. Jaffe. More Differential Power Analysis: Selected DPA Attacks, June 2006. Presented at ECRYPT Summerschool on Cryptographic Hardware, Side Channel and Fault Analysis.
8. M. Joye, P. Paillier, and B. Schoenmakers. On Second-Order Differential Power Analysis. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 293–308. Springer, 2005.

9. P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.
10. S. Mangard. A Simple Power-Analysis (SPA) Attack on Implementations of the AES Key Expansion. In P. J. Lee and C. H. Lim, editors, *Information Security and Cryptology - ICISC 2002, 5th International Conference Seoul, Korea, November 28-29, 2002, Revised Papers*, volume 2587 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2003.
11. S. Mangard. Hardware Countermeasures against DPA – A Statistical Analysis of Their Effectiveness. In T. Okamoto, editor, *Topics in Cryptology - CT-RSA 2004, The Cryptographers' Track at the RSA Conference 2004, San Francisco, CA, USA, February 23-27, 2004, Proceedings*, volume 2964 of *Lecture Notes in Computer Science*, pages 222–235. Springer, 2004.
12. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks – Revealing the Secrets of Smart Cards*. ADIS Book Series. Springer, 2007. ISBN 0-387-30857-1.
13. E. Oswald and S. Mangard. Template Attacks on Masking—Resistance is Futile. In M. Abe, editor, *Topics in Cryptology - CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007, Proceedings*, volume 4377 of *Lecture Notes in Computer Science*, pages 243–256. Springer, February 2007.
14. E. Oswald, S. Mangard, C. Herbst, and S. Tillich. Practical Second-Order DPA Attacks for Masked Smart Card Implementations of Block Ciphers. In D. Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*, pages 192–207. Springer, 2006.
15. E. Oswald, S. Mangard, N. Pramstaller, and V. Rijmen. A Side-Channel Analysis Resistant Description of the AES S-box. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption, 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, volume 3557 of *Lecture Notes in Computer Science*, pages 413–423. Springer, 2005.
16. E. Oswald and K. Schramm. An Efficient Masking Scheme for AES Software Implementations. In J. Song, T. Kwon, and M. Yung, editors, *Information Security Applications, 6th International Workshop, WISA 2005, Jeju Island, Korea, August 22-24, 2005, Revised Selected Papers*, volume 3786 of *Lecture Notes in Computer Science*, pages 292–305. Springer, 2006.
17. F.-X. Standaert, E. Peeters, and J.-J. Quisquater. On the Masking Countermeasure and Higher-Order Power Analysis Attacks. In *International Symposium on Information Technology: Coding and Computing (ITCC 2005), 4-6 April 2005, Las Vegas, Nevada, USA, Proceedings*, volume 1, pages 562–567. IEEE Computer Society, April 2005.
18. S. Tillich and J. Großschädl. Instruction Set Extensions for Efficient AES Implementation on 32-bit Processors. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems – CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 270–284. Springer, 2006.
19. J. Waddle and D. Wagner. Towards Efficient Second-Order Power Analysis. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems – CHES 2004, 6th International Workshop, Cambridge, MA, USA, August 11-13, 2004, Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2004.