Randomizing the Montgomery Multiplication to Repel Template Attacks on Multiplicative Masking

Marcel Medwed and Christoph Herbst

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

Abstract. For a long time multiplicative masking together with highly regular exponentiation algorithms was believed to thwart all side-channel based threats. Recent research results showed that the multiplicative masking itself can be attacked in order to recover the used masks. In this paper we propose a countermeasure which closes this security gap. The basic idea is to protect the masking step by introducing a randomized multiplication. The proposed method is cheap in terms of performance overhead. The memory overhead is reasonable.

Keywords: RSA, Side-Channel Analysis, Countermeasure, Template Attacks, Randomized Montgomery Multiplication, Multiplicative Masking

1 Introduction

Nowadays, electronic processing and transmission of data is indispensable. If the transmitted data contains sensitive information, security issues arise. Depending on the application various security services are needed. Those are mainly proving the authenticity of data or parties involved in the communication, data integrity, and the secrecy of the data. Driven by the advances in microelectronics and network technology even restricted devices like embedded systems are part of such communication networks.

A preferable tool to achieve authenticity, integrity, and secrecy is cryptography. Based on cryptographic primitives, protocols can be designed and implemented to achieve the required properties. The security of cryptographic algorithms relies on the fact that the so-called secret key is only known by authorized entities. Furthermore, the security is based on the assumption, that it is not possible to derive or extract the key by analyzing the relationship between plaintexts and corresponding ciphertexts.

Even if used algorithms like RSA, DSA, ECDSA, AES, or similar are believed to be secure against cryptanalysis, their implementations can be insecure. Attacks which aim on the implementation of an algorithm rather than on the algorithm itself are so-called *implementation attacks*. These attacks make use of weaknesses of the device or the implementation inside the device. They try to gather additional information, leaked by the device. This information is then used to extract the key processed by the attacked device. There are several ways an attacker can obtain such information. For instance the adversary can induce a fault into the device in order to disturb the computation of the result. If such a faulty result is used to recover the secret the notion *fault attack* is used. First results have been shown by Boneh et al. [2] and Biham and Shamir [1] in 1997. Another possibility is to observe information which is emitted physically by the device. The usage of such data to gain knowledge about the internal states of the device is denoted as *side-channel attacks* or Side-Channel Analysis (SCA) attacks. They have been proposed by Kocher et al. in [8] and [9]. Examples for measurable side-channel information are the power consumption, electromagnetic emanation, or timing information. In [8], Kocher proposed timing attacks, whereas the topic of [9] is power analysis attacks.

To perform power analysis attacks, the power consumption of the device under attack is measured while the desired secret key is processed. In power analysis attacks we distinguish between Simple Power Analysis (SPA), where the secret key is extracted with the help of a single measurement trace (power trace), and Differential Power Analysis (DPA) attacks, where multiple traces with different input values have to be acquired. First results of power analysis attacks on asymmetric algorithms like RSA [15] have been published by Messerges et al. [12].

The most powerful SPA attacks are template attacks, which haven been introduced by Chari et al. [5]. Template attacks are two-stage attacks. In the first phase, an attacker uses a similar device which he can fully control, to model the power consumption of the device under attack. He builds so-called templates for each possible data value of a certain operation he wants to attack. In the second phase of the attack one power trace is measured during the computation and then matched against the built templates. The template which fits best indicates the data value used in the measured operation with a high probability. First practical results have been published in [14]. In [11] template attacks on ECDSA have been presented.

Beside new and better attacks, international research also focused on the design and implementation of countermeasures against SCA attacks. In general, there are two groups of countermeasures, namely masking and hiding [10]. Masking brakes the link between the predicted intermediate values and the actually occurring intermediates in the implementation, whereas hiding minimizes the effect of the processed data on the power consumption of the device. Algorithmic countermeasures for RSA have already been published in [8]. In [6], Fumarolli et al. presented a blinded and fault resistant implementation of RSA using a Montgomery exponentiation ladder implementation. Most published side-channel analysis countermeasures for asymmetric algorithms are based on multiplicative masking.

Until [7], multiplicative masking was believed to be secure. In this article it is shown that by using template attacks and a sieving step, the random mask used for multiplicative masking could be extracted. Subsequently, the intermediate values could be unmasked. Their attack is limited to processors with word sizes up to 16 bit. In this paper we will propose a randomized Montgomery multiplication to repel this attack. We will furthermore present a performance-optimized software implementation of the general idea for an 8-bit platform. In addition, we will give a security analysis and we show that the overall performance of a blinded RSA implementation will not suffer by using a randomized multiplication.

The remainder of this paper is organized as follows. Section 2 revisits the attack on the multiplicative masking. In Section 3, the general idea of the countermeasure is explained. The application of the idea to a Montgomery CIOS (Coarsely Integrated Operand Scan) multiplication is described in Section 4. We evaluate the security of our proposed countermeasure in Section 5. Finally, in Section 6 conclusions are drawn.

2 Attack

Public key algorithms often use multiplicative masking as a protection against side-channel attacks. Therefore, the message is multiplied by a random mask in the beginning of the encryption. At the end, the result is divided by a value related to the mask, thus the result is unmasked again. As a consequence, the processed intermediate values seem all random to the observer who mounts a side-channel attack. However, the multiplication which is needed for the initial masking is not protected. The attack proposed in [7] aims at this initial masking. The attack allows to recover the used random value after observing the masking operation and hence to unmask the whole computation.

The attack itself can be divided into two phases: The observing phase and the sieving phase. In the observing phase, the power consumption during the long integer multiplication of the masking is measured. From the measured power consumption, information about the mask can be derived. The mask is usually longer than the word size of the device under attack. Hence, the mask consists of several words. Every time one of these words is processed, the power consumption of the processor contains information about the Hamming weight of the word. The Hamming weight itself is extracted from the power traces using template attacks. Thus, the result of the observing phase is the knowledge about the Hamming weights of the words which are directly or indirectly related to the unknown mask. The knowledge of these Hamming weights is then used to recover the exact values of the processed words in the sieving phase. This second phase filters out impossible candidates for the mask. If enough information is available during this sieving phase then the possible mask space can be narrowed down to one single candidate.

Before looking at the sieving step, we need to know how to gather enough information for a successful sieving. Therefore, we revise a standard long integer multiplication algorithm and template attacks. After reviewing the sieving step, we discuss the feasibility of the attack.

2.1 Long Integer Multiplication

A typical long-integer multiplication algorithm can be seen in figure 1. This so-called text-book method takes every word of operand A and multiplies it by every word of operand B. The result of such a single precision multiplication is added to the appropriate digit of the overall result. What is vital for the attack is that the words of the operands and the results of the single precision multiplications processed separately. Furthermore, it is advantageous that those results are processed in two parts, the high word C and the low word T. It is also important that the attacker knows exactly which algorithm is used and when which word is processed.

Algorithm 1 Multi-precision integer multiplication

```
Require: Multi-precision integers $A, B$ consisting of $l$ words
Ensure: $T = A \cdot B$
 1: $T \leftarrow 0$
 2: for $i$ from 0 to $l-1$ do
 3:     $C \leftarrow 0$
 4:     for $j$ from 0 to $l-1$ do
 5:         $(CS) \leftarrow T[i+j] + A[i] \cdot B[j] + C$
 6:         $T[i+j] \leftarrow S$
 7:     end for
 8:     $T[i+l] \leftarrow C$
 9: end for
10: return $T$
```

2.2 Template Attacks

Template attacks present the most powerful simple power analysis techniques. As well as other power analysis techniques like DPA, they are based on statistics. A template represents a statistical model for the power consumption of a given operation. It is assumed that the power consumption of a device follows a multivariate normal distribution (MVN). Such a distribution can be fully described by a mean vector and a covariance matrix. Hence, looking at the points in a power trace and their dependencies amongst each other, the power consumption of a certain operation can be described by the mean vector $\mathbf{m}$ and the covariance matrix $\mathbf{C}$. A template is such a pair $(\mathbf{m}, \mathbf{C})$. However, in order to build such a model the attacker needs a training device which behaves similarly to the device under attack. Under this assumption the adversary can for instance model the power consumption of all nine MOV instructions with different Hamming weights on an 8-bit processor.

Since the adversary also knows the moments of time when which values are processed within the power trace, he can extract those points and apply the previously built templates to them. That is, for sets of interesting points, where a relevant Hamming weight can be observed, all built templates are matched against. Matching means that the probability density function (PDF) is evaluated with a template and the interesting points as input. For every template the result of the matching is a probability value stating how likely it is that the modeled operation/Hamming weight really occurred. According to the maximum-likelihood decision-rule it is assumed that the operation with the highest probability was actually performed.

After matching the templates against all interesting points, we know the Hamming weights of the operands as well as those of the partial products of the multiplication.

2.3 Sieving Step

The sieving step uses the previously gathered information about the occurring Hamming weights and tries to narrow down the possible mask candidates. It is assumed that only one operand is secret, the other one, for instance the message, is known.

In the first phase of the sieving, the initially possible candidates for the unknown input words $A[i]$ are determined. For instance, if a Hamming weight of two was observed for $A[0]$ then there are only 28 values in question for this word.

The second phase looks at the partial products. For every left candidate of $A[0]$, the hypothetical partial products with all words of B are computed. Afterwards it is checked which candidate leads to the Hamming weights extracted from the power trace. If a candidate leads to a partial-product hypothesis which is not possible according to the extracted Hamming weights, the candidate is dismissed.

The success rate of the sieving step increases with the operand length. It has been shown in [7] that for common RSA operands, the recovery of the mask is feasible. The attack basically does not depend on the word size of the architecture. That is, the success rates are the same for 8- and 16-bit platforms However, for 32-bit platforms the sieving step becomes computationally infeasible.

The maybe most surprising outcome of the paper is that the templates do not need to be exact. That is, if a template indicates a Hamming weight different from the occurring Hamming weight, this can be considered and tolerated if the error stays within a certain range. It turned out that the attack works for 8-bit platforms up to a tolerance of four. Hence, the attack also works in slightly noisy environments.

3 Randomized Montgomery Multiplication

The general idea of the Randomized Montgomery Multiplication Countermeasure (RMMC) is to randomize the calculation of the product. More precisely, the single partial products are calculated at different points in time for each secured multi-precision multiplication. Even the summation of the different partial products will take place in randomly changing order. Algorithm 1 gives the pseudocode for a traditional text-book method, whereas Algorithm 2 is the randomized version. Again, it is assumed that A holds the random mask and that the known value (e.g. message) is processed in the outer (not randomized) loop. The randomization is based on at least one randomization vector determining which partial product is calculated in the actual iteration of the inner loop. Such randomization vectors can be generated in linear time by swapping randomly indexed elements of a sequence. In Section 5 we will show why it can be necessary to use a different randomization vector for each iteration of the outer loop.

The most obvious difference between the traditional algorithm and the randomized version is the fact that the calculation of the partial products and their summation is split up into two parts in the randomized version. This is due to the fact that the partial products are calculated in random order and hence, the algorithm can not determine if the corresponding carry is already available. Therefore, the first inner loop (lines 4 to 7 of Algorithm 2) calculates the par-

Algorithm 2 Randomized multi-precision integer multiplication

Require: Multi-precision integers A, B, where A consists of $l+1$ and B of l words, and a randomization vector rv consisting of a permutation of the values 0 to l. The word of A with the index l is zero; A carry array C with $(l+2)$ elements and an array T with $(2l+1)$ elements.

Ensure: $T = A \cdot B$

```
 1: for $i$ from 0 to $l-1$ do
 2:   $T \leftarrow 0$
 3:   $C \leftarrow 0$
 4:   for $j$ from 0 to $l$ do
 5:     $index \leftarrow rv[j]$
 6:     $(C[index], T[index+i]) \leftarrow T[index+i] + A[index] \cdot B[i]$
 7:   end for
 8:   for $u$ from 0 to $l-1$ do
 9:     for $j$ from 0 to $l$ do
10:       $index \leftarrow rv[j]$
11:       $(X, T[index+i+1]) \leftarrow T[index+i+1] + C[index]$
12:       $C[index] \leftarrow 0$
13:       $C[index+1] \leftarrow C[index+1] + X$
14:     end for
15:   end for
16: end for
17: return $T$
```

tial products of $A[index] \cdot B[i]$ where $index$ is the value of the randomization vector rv at position j. The high word of the result is stored in the carry vector C and the low word in the accumulated result T. The second big difference is that the accumulation of the partial products has to be done in a nested loop (lines 8 to 15 of Algorithm 2). The reason for this is the fact that because of the random order of the summation it has to be made sure that the carry can propagate from the lowest word to the highest word. The carry propagation must be performed in randomized order. Otherwise, an attacker could extract the order of the randomized multiplications or at least minimize the possibilities. If the carry is propagated in a straight-forward way, an observer can learn the Hamming weights from the single carry bytes. Now he can try to use this information to reorder the partial products according to this information. For the worst case this carry propagation can only be ensured by performing the summation step $l * (l - 1)$ times.

Why does the randomization now defeat a template attack on the multiplication? For the sieving step it is essential that the attacker knows which element has been multiplied with which one. Otherwise the sieving cannot be successful. One possibility for an attacker would now be to guess the randomization. With a probability of $1/(l!)$ this would succeed but only if one randomization vector is used for the entire multiplication. If a different randomization vector is used for each iteration of the outer loop, the probability of guessing the correct randomization vectors for all rounds will drop to $(l!)^{-l}$. Using different vectors also helps to defeat an attack which can be a threat to the randomized multiplication algorithm when applied to the CIOS version of the Montgomery multiplication. This attack is sketched in Section 5.

The stated probabilities of $l!$ and $(l!)^{-l}$ are the maximum values if no additional information about the randomization vectors or the mask is available. Such additional information is the sequence of Hamming weights of the randomization vector. Also knowing the Hamming weights of the mask in correct order provides such an additional information.

4 Protected CIOS

After presenting the basic concept of our countermeasure in Section 3 we will show how to apply the general idea to an implementation of the so-called Montgomery multiplication algorithm due to Peter L. Montgomery [13] in this section. We have decided to implement a special version of the multiplication, namely the CIOS. The CIOS method has been analyzed in [4] concerning its timing and memory requirements. Based on this analysis the CIOS method seems to be a good choice for implementations on restricted devices like 8-bit or 16-bit microcontrollers. Amongst the compared methods CIOS has the highest performance while the RAM usage is kept small. Another reason why we chose to implement a CIOS multiplication is that our randomization approach can be integrated easily. For more details on CIOS see [4].

Algorithm 3 Randomized Montgomery CIOS Multiplication

```
Require: Two multi-precision integers $A$, $B$ with $l$ $b$-bit words; the modulus $N$;
    $n' = -N^{-1} \pmod W$; an array $r1$ consisting of $l$ elements which are the permutation of the numbers $0\ to\ (l-1)$; an array $r2$ consisting of $l+2$ elements which are a permutation of the numbers $0\ to\ (l+1)$; A carry array $C$ with $(l+3)$ elements and an array $T$ with $(l+2)$ elements; $W$ is $2^b$.
Ensure: $T = A \cdot B \pmod N$
 1: for $i$ from 0 to $(l-1)$ do
 2:   $T \leftarrow 0$
 3:   $C \leftarrow 0$
 4:   for $j$ from 0 to $(l-1)$ do
 5:     $index1 = r1[j]$
 6:     $(C[index1+1], T[index1]) = T[index1] + A[index1] \cdot B[i]$
 7:   end for
 8:   for $u$ from 0 to $(l+1)$ do
 9:     for $j$ from 0 to $(l+1)$ do
10:       $index2 = r2[j]$
11:       $(X, T[index2]) = T[index2] + C[index2]$
12:       $C[index2] = 0$
13:       $C[index2+1] = C[index2+1] + X$
14:     end for
15:   end for
16:   $C \leftarrow 0$
17:   $m = T[0] \cdot n'[0]\ mod\ W$
18:   for $j$ from 0 to $(l-1)$ do
19:     $index1 = r1[j]$
20:     $(C[index1+1], T[index1]) = T[index1] + N[index1] \cdot m$
21:   end for
22:   for $u$ from 0 to $(l+1)$ do
23:     for $j$ from 0 to $(l+1)$ do
24:       $index2 = r2[j]$
25:       $(X, T[index2]) = T[index2] + C[index2]$
26:       $C[index2] = 0$
27:       $C[index2+1] = C[index2+1] + X$
28:     end for
29:   end for
30:   for $j$ from 0 to $(l+1)$ do
31:     $index2 = r2[j]$
32:     $C[index2] = T[index2]$
33:   end for
34:   for $j$ from 0 to $(l+1)$ do
35:     $index2 = r2[j]$
36:     $T[index2] = C[index2+1]$
37:   end for
38: end for
39: Final_Compare_and_Subtract($T$,$N$);
```

In Algorithm 3, a straightforward integration into a CIOS multiplication of the concept described in Section 3 is given. The randomization is ensured via the randomization vectors $r1$ and $r2$. The elements of the vectors are a permutation of the numbers 0 to $l-1$ and 0 to $l+1$ respectively. We are randomizing the partial multiplications which take place in the loop performed in lines 4 to 7. This is done by using the entry at position j in the randomization vector $r1$ to index the elements in A, T, and C. As already mentioned in Section 3 the multiplication and the summation have to be split up in two loops, because due to the randomization one can not be sure that the preceding partial product has already been calculated. Therefore, the carry which has to be added to the actual partial product is unknown. The summation of the carry takes place in the lines 8 to 15. Due to the fact that here more than l elements have to be added, another randomization vector is needed. The calculation of the partial products in CIOS is immediately followed by a reduction step.

This reduction step is a simple addition of T and the product $m \cdot N$. The value m is calculated in line 17. This partial multiplication of $m \cdot N$ and addition to T is performed randomized in the lines 18 to 21 of Algorithm 3. The summation of the carries is randomized in the nested loops in lines 22 to 29. The final step of the inner loop of the CIOS method is the shifting of the accumulated result T by one word to the right. If performed in a randomized way this has to be done in two steps, which are given in the lines 30 to 37.

When analyzing this implementation it is immediately visible that the performance scales with l^3. Compared to CIOS which scales with l^2 this leads to a big loss in performance. Furthermore, one can see that two separated randomization vectors are necessary and that the high words of each partial product have to be stored. A more detailed performance and memory comparisons are given later on. To overcome the performance gap between an unprotected and a protected implementation we have designed a second algorithm which uses a carry-save strategy in software. This allows to omit the nested loops. Algorithm 4 gives the implementation of our proposed countermeasure in a more sophisticated way. The main differences to Algorithm 3 are the additional second carry vector, the absence of a second randomization vector, and the removal of the nested loop. However, to achieve this better scalability we have to spend memory for the realization of the carry-save strategy. We use two carry vectors and swap them just before performing the partial multiplications in the lines 5 and 12. This carry-save representation of the numbers has to be resolved at the end of the algorithm (lines 26 to 29 of Algorithm 4). A further tweak is realized in line 15 of Algorithm 4. Compared to line 8 the high word of the result is not written to the next position of the carry vector. This can be done because in the lines 17 to 24 the accumulated result T is shifted one position to the right. To save this shifting step for the carry we write it to the correct position immediately. It turns out that the invested memory not only increases the speed but also the second randomization vector is obsolete and therefore no additional memory is used.

Table 1 gives the performance and memory figures for our two proposed algorithms when implemented in assembly on an 8-bit ATmega163 microcontroller.

Algorithm 4 Randomized Montgomery CIOS Multiplication with Carry Save

```
Require: Two multi precision integers $A$,$B$ with $l$ $b$-bit words; the modulus $N$;
    $n' = -N^{-1} \pmod{W}$;the variables $A$ and $N$ consist of $l+1$ words where the
    most significant word is zero; an array $r3$ consisting of $l+1$ elements which are a
    permutation of the numbers $0\,to\,l$; Two carry arrays ($CI$;$CO$) with $(l+2)$ elements
    and a result array $T$ with $(l+2)$ elements, $W$ is $2^b$.
Ensure:
 1: $T \leftarrow 0$
 2: $CI \leftarrow 0$
 3: $CO \leftarrow 0$
 4: for $i$ from 0 to $(l-1)$ do
 5:     swap($CI$,$CO$)
 6:     for $j$ from 0 to $l$ do
 7:         $index3 = r3[j]$
 8:         $(CO[index3+1], T[index3]) = T[index3] + A[index3] \cdot B[i] + CI[index3]$
 9:     end for
10:     refresh(r3)
11:     $m = T[0] \cdot n'[0] \pmod{W}$
12:     swap($CI$,$CO$)
13:     for $j$ from 0 to $(l)$ do
14:         $index3 = r3[j]$
15:         $(CO[index3], T[index3]) = T[index3] + N[index3] \cdot m + CI[index3]$
16:     end for
17:     for $j$ from 0 to $l$ do
18:         $index3 = r3[j]$
19:         $CI[index3+1] = T[index3+1]$ // Note $T[0]$ is dismissed because it is 0
20:     end for
21:     for $j$ from 0 to $l$ do
22:         $index3 = r3[j]$
23:         $T[index3] = CI[index3+1]$
24:     end for
25: end for
26: $temp\_carry = 0$
27: for $j$ from 0 to $(l+1)$ do
28:     $(temp\_carry, T[j]) = T[j] + CO[j] + temp\_carry$
29: end for
30: Final_Compare_and_Subtract($T$,$N$);
```

The performance and memory figures of an assembly implementation of the plain CIOS multiplication method on the same platform are given for comparison. The memory requirements are measured in a multiple of the number of words denoted by l. The performance values for the randomized CIOS multiplication highlight the bad scaling factor of the performance for increasing operand sizes. For the randomized version of the CIOS multiplication using the carry-save strategy performance is scaled down by a factor of 3 in relation to the plain CIOS implementation. Randomization not only decreases the performance but also increases the memory usage. Compared to the non-randomized implementation we need $3l$ words of additional memory for both implementations.

Algorithm 5 Blinded Fault-Resistant Montgomery ladder by Fumaroli et al. [6]

```
Require: $x \in \mathbb{G}, k = \sum_{i=0}^{t-1} k_i 2^i \in \mathbb{N}$
Ensure: $x^k \in \mathbb{G}$
 1: Pick a random $r \in \mathbb{G}$
 2: $R_0 \leftarrow r$; $R_1 \leftarrow rx$; $R_2 \leftarrow r^{-1}$
 3: init(CKS)
 4: for $j = t-1$ down to 0 do
 5:     $R_{\neg k_j} \leftarrow R_{\neg k_j} R_{k_j}$
 6:     $R_{k_j} \leftarrow R_{k_j}^2$
 7:     $R_2 \leftarrow R_2^2$
 8:     update(CKS,$k_j$)
 9: end for
10: $R_2 \leftarrow R_2 \oplus CKS \oplus CKS_{ref}$
11: return $R_2 R_0$
```

Algorithm 6 Blinded Fault-Resistant Exponentiation by Boscher et al. [3]

```
Require: $x \in \mathbb{G}, k = \sum_{i=0}^{t-1} k_i 2^i \in \mathbb{N}$
Ensure: $x^k \in \mathbb{G}$
 1: Pick a random $r \in \mathbb{G}$
 2: $R_0 \leftarrow r$; $R_1 \leftarrow r^{-1}$; $R_2 \leftarrow x$
 3: for $j = 0$ to $t-1$ do
 4:     $R_{\neg k_j} \leftarrow R_{\neg k_j} R_2$
 5:     $R_2 \leftarrow R_2^2$
 6: end for
 7: if $(R_1 R_2 x = R_2)$ then then
 8:     return $(r^{-1} R_0)$
 9: else
10:     return ("Error")
11: end if
```

The loss of performance is immediately put into perspective when regarding a secured RSA implementation, e.g. masked Montgomery ladder. As explained in [7] the weak points of multiplicative masking as security measure are the masking step and the mask-update step. Our randomized multiplication can be used in these steps to secure the multiplication. Furthermore, the Montgomery conversion of the mask has to be protected. This multiplication can also be performed with our randomized multiplication algorithm. All other multiplications during the calculation of the RSA algorithm are then protected by the multiplicative masking. Calculating an RSA encryption with a 512-bit exponent using Algorithm 5 takes roughly $1,550$ multiplications. When using our randomized carry-save approach one multiplication takes 3 times longer. But not all mask updates (line 7) have to be protected for RSA. Since calculating the square root is not possible modulo n, we only need to protect a reasonable amount of mask squarings in the beginning. This is because an attacker who recovers $R_2^{2^{t-1-j}}$ can guess the already processed key and thus recover a small set of possible intermediate values. The intermediate values can then be used to mount a template attack like in [11]. Thus, roughly 40 out of the $1,550$ Montgomery multiplications have to be replaced. For the complete RSA, this results in a performance overhead of about 5% percent or even less for larger exponents as the overhead halves as the keylength doubles. Note that all mask updates have to be protected if calculating square roots is possible, like for ElGamal. This would result in a constant overhead of approximately 65%.

Recently, Boscher et. al [3] introduced another exponentiation algorithm which addresses side-channel and fault issues. This algorithm (Algorithm 6) needs only two multiplications per exponent bit. If we want to apply our countermeasure to this algorithm, we need to address the lines 2 and 4. Line 2 needs to be protected because the authors propose to generate r and its inverse by means of exponentiation with a small exponent. We assume 80 multiplications here, for more, using the extended GCD algorithm is likely to be the cheaper solution. The problem with line 4 is that the attack can be used to recover $x^{k'}r$, where k' is the already processed part of k (or the bit-complement, depending on k_j). This is because R_2 can be assumed as known (via the timing). If j is close enough to 0, there are 2^j possible values for $x^{k'}$ and each of them yields a different mask. In a next step a template attack like in [11] can be mounted for every mask. The attack works analog if j is close to $t-1$. As a consequence the algorithm has to be protected during the first few and the last few bits of k. Assuming that it is impossible to follow the sketched approach for more than 40 bits, it suffices to protect 80 multiplications in line 4 and another 80 in line 2. Hence, we have to protect 160 out of (1024+80) multiplications which results in an overhead of roughly 29%. Again, the overhead halves for double the bitlength of k. For Boscher et. al.'s algorithm, this variable overhead would also apply for ElGamal.

Depending on the exponentiation algorithm and the public knowledge about the underlying group, the overhead can be non-negligible, especially for embedded systems. However, if the protection profile requires security against template attacks, there is (up to our best knowledge) no solution which is less performance-intensive per multiplication. Moreover, we have to bear in mind that this is a software countermeasure which can be applied to all processor-driven devices and that no dedicated hardware is necessary.

Table 1. Performance results on an 8-bit ATmega163 microcontroller for our randomized CIOS implementation in comparison to an optimized CIOS multiplication

Implementation	**l = 64** Cycles	**l = 32** Cycles	**l = 16** Cycles	**Memory** Words
CIOS multiplication	149,934	38,190	9,966	$4l+10$
Randomized CIOS multiplication	14,395,427	1,966,243	289,763	$7l+10$
Randomized CIOS multiplication with carry save	438,350	112,846	29,966	$7l+11$

5 Security Evaluation

Randomization in time is a well known countermeasure for DPA attacks. There, the correlation is reduced by a factor of R which is identical to the randomization degree. If strong attacking techniques like windowing are applied [16], the gain is reduced to $\sqrt{R}$. However, it is important to consider the difference between a DPA attack scenario and a template attack scenario. In our scenario it is vital for the attacker to know the correct order in which the operand is processed. This is because the attacker is only given a single trace and the attack fails immediately if a correct candidate is removed from the set of possible candidates.

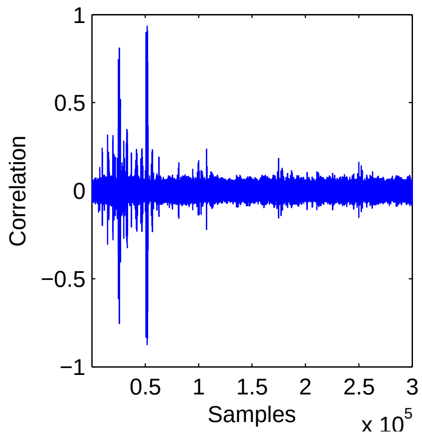

Fig. 1. Result of DPA attack on the lower byte of $A[1] \cdot B[0]$ for 1,000 traces without randomization

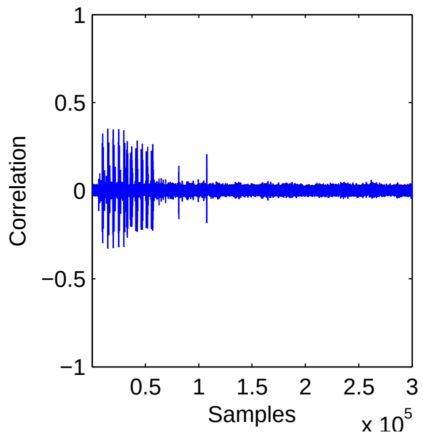

Fig. 2. Result of DPA attack on the lower byte of $A[1] \cdot B[0]$ for 5,000 traces with randomization

To show the effectiveness of the randomization we performed a DPA attack on the partial products $A[1] \cdot B[0]$. It can be seen that there are two DPA peaks when the words of operand A are not randomized (Figure 1). The other small peaks arise due to the fact that the partial products are not completely independent. This is because they depend on the same word of B. However, as soon as randomization is introduced, the peaks decrease by the number of words l in A (see Figure 2, with $l = 4$). This does not directly thwart the template attack. However, if the partial products are randomized, the matching step of the template attack fails.

Nevertheless, looking at the partial product of $A[0] \cdot B[0]$ in Figure 3 and Figure 4 an additional peak occurs which even remains in the randomized setup. This peak is caused by the code in line 11 of Algorithm 4. Here, the lowest word of the result (the low-byte of $A[0] \cdot B[0]$) is accessed to calculate the summand for the reduction. This part is hard to randomize, except with the addition of dummy cycles. However, what is worse is that it poses a threat to the whole randomization. If the same permutation vector is used for all rounds, the following attack would work to recover the permutation sequence: The result used for line 11 also occurs in the multiplication above. Hence, it matches with a subset of the n positions for the partial products. By performing this matching for several rounds of the outer loop, the position where $A[0] \cdot B[i]$ is processed can be recovered. Now that this position is known, the position of $A[1] \cdot B[i]$ can be

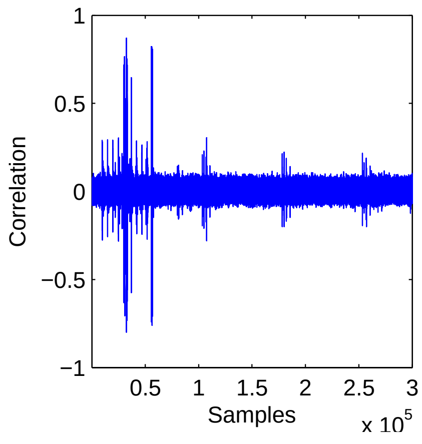

Fig. 3. Result of DPA attack on the lower byte of $A[0] \cdot B[0]$ for 1,000 traces without randomization

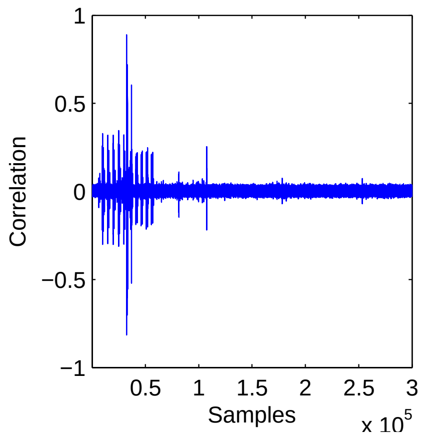

Fig. 4. Result of DPA attack on the lower byte of $A[0] \cdot B[0]$ for 5,000 traces with randomization

attacked following a similar strategy. The link between $A[0] \cdot B[i]$ and $A[1] \cdot B[i]$ is the carry byte. Hence, the full randomization vector can be obtained. However, this can be prevented if every round is randomized with another permutation vector.

6 Conclusion

In this article we presented a method to protect multiplicative masking. This approach allows to repel for instance the attack described in [7]. Our proposed countermeasure is based on a randomized multiplication. This is because randomization is effective against single-shot attacks where knowledge about the order of the operations is vital for their success. Besides the general idea of the countermeasure we presented a straightforward implementation for an 8-bit ATmega163 microcontroller. We also showed how to improve the performance of the basic realization. This was done by using a carry-save representation for the intermediate results. It turned out that such an approach increases the runtime by a constant factor of about 3. However, only a fraction of the multiplications for a protected RSA has to be replaced. Thus the overhead is below 5% for Fumaroli et. al.'s algorithm. For Boscher et. al.'s algorithm, the overhead can stated with below 30%. The memory requirements increase by $3l$, where l is the number of words of the message. In order to overcome the countermeasure and to recover the mask, an adversary would need to guess the used randomization vectors. This would require an effort of $l!^l$. The proposed countermeasure is intended to close the security gap for protected RSA implementations presented in [7].

Acknowledgements. The work described in this paper has been supported in part through the Austrian Government funded projects *ARTEUS* and *POWER-TRUST* established under the Trust in IT Systems program FIT-IT. 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. 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.
2. 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.
3. A. Boscher, H. Handschuh and E. Trichina. Blinded Fault Resistant Exponentiation Revisited. In L. Breveglieri, I. Koren, D. Naccache, E. Oswald and J.-P. Seifert, editors, *Fault Diagnosis and Tolerance in Cryptography, Sixth International Workshop, FDTC 2009, Lausanne, Switzerland, September 6, 2009, Proceedings*, pages 3–9.
4. Çetin Kaya Koç, T. Acar, and B. S. K. Jr. Analyzing and Comparing Montgomery Multiplication Algorithms. *IEEE Micro*, 16(3):26–33, June 1996.
5. S. Chari, J. R. Rao, and P. Rohatgi. Template Attacks. In B. S. K. Jr., Çetin Kaya Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2003.
6. G. Fumaroli and D. Vigilant. Blinded Fault Resistant Exponentiation. In L. Breveglieri, I. Koren, D. Naccache, and J.-P. Seifert, editors, *Fault Diagnosis and Tolerance in Cryptography, Third International Workshop, FDTC 2006, Yokohama, Japan, October 10, 2006, Proceedings*, volume 4236 of *Lecture Notes in Computer Science*, pages 62–70. Springer, October 2006.
7. C. Herbst and M. Medwed. Using Templates to Attack Masked Montgomery Ladder Implementations of Modular Exponentiation. In K.-I. Chung, M. Yung, and K. Sohn, editors, *9th International Workshop on Information Security Applications (WISA 2008), Jeju Island, Korea, September 23-25, 2008, Pre-Proceedings*, 2008.
8. P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In N. Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, number 1109 in Lecture Notes in Computer Science, pages 104–113. Springer, 1996.
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, E. Oswald, and T. Popp. *Power Analysis Attacks – Revealing the Secrets of Smart Cards*. Springer, 2007. ISBN 978-0-387-30857-9.
11. M. Medwed and E. Oswald. Template Attacks on ECDSA. In K.-I. Chung, M. Yung, and K. Sohn, editors, *9th International Workshop on Information Security Applications (WISA 2008), Jeju Island, Korea, September 23-25, 2008, Pre-Proceedings*, 2008.
12. T. S. Messerges, E. A. Dabbish, and R. H. Sloan. Power Analysis Attacks of Modular Exponentiation in Smartcards. In Çetin Kaya Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES'99, First International Workshop, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 144–157. Springer, 1999.
13. P. L. Montgomery. Modular Multiplication without Trial Division. *Mathematics of Computation*, 44:519–521, 1985.
14. C. Rechberger and E. Oswald. Practical Template Attacks. In C. H. Lim and M. Yung, editors, *Information Security Applications, 5th International Workshop, WISA 2004, Jeju Island, Korea, August 23-25, 2004, Revised Selected Papers*, volume 3325 of *Lecture Notes in Computer Science*, pages 443–457. Springer, 2004.
15. R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978. ISSN 0001-0782.
16. S. Tillich and C. Herbst. Attacking State-of-the-Art Software Countermeasures—A Case Study for AES. In E. Oswald and P. Rohatgi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2008, 10th International Workshop, Washington DC, USA, August 10-13, 2008, Proceedings*, volume 5154 of *Lecture Notes in Computer Science*, pages 228–243. Springer, August 2008.