Reduce-by-Feedback: Timing resistant and DPA-aware Modular Multiplication plus: How to Break RSA by DPA

Michael Vielhaber

Hochschule Bremerhaven, FB2, An der Karlstadt 8, D–27568 Bremerhaven, Germany
Universidad Austral de Chile, Instituto de Matemáticas, Casilla 567, Valdivia, Chile
vielhaber@gmail.com

Abstract. We (re-) introduce the Reduce-By-Feedback scheme given by Vielhaber (1987), Benaloh and Dai (1995), and Jeong and Burleson (1997).
We show, how to break RSA, when implemented with the standard version of Reduce-by-Feedback or Montgomery multiplication, by Differential Power Analysis. We then modify Reduce-by-Feedback to avoid this attack. The modification is not possible for Montgomery multiplication. We show that both the original and the modified Reduce-by-Feedback algorithm resist timing attacks.
Furthermore, some VLSI-specific implementation details (delayed carry adder, re-use of MUX tree and logic) are provided.
Keywords: Reduce-by-Feedback, modular multiplication, Montgomery multiplication, timing analysis, differential power analysis.

1 Introduction

RSA, Diffie-Hellman (over $\mathbb{F}_p$), and elliptic curve schemes (over $\mathbb{F}_p$) use modular multiplication as their computational kernel. This is usually implemented as Montgomery multiplication [12] (1985), which is fast and has timing independent of the values. Montgomery treats the bits of the first factor to be multiplied from the LSB towards the left, and works with the residue classes $[x \cdot (2^L)^{-1}] \mod N$, where $[x]$ are the standard residue classes, and L is the length (in bits) of the operands, *e.g.* $L = \lceil \log_2(N) \rceil$.

There exists, however, an algorithm that avoids the mapping from $[x]$ to $[x \cdot (2^L)^{-1}] \mod N$, by working the bits of the first factor from MSB downwards to the right: Reduce-by-Feedback [15, 16, 20] (1987) (Sections 3 and 4).

The Reduce-by-Feedback algorithm preserves the immunity against timing attacks (Section 5), the constant shift amount of 1,2,3, or 4 bits per clock cycle, depending on the implementation effort, and all other advantages of Montgomery multiplication.

Additionally, a DPA attack against RSA implemented by Montgomery multiplication or Reduce-by-Feedback (Section 6), can be avoided by a modification

of Reduce-by-Feedback (Section 7). This modification can not be applied to Montgomery multiplication, as far as we can see.

An overview about implementations of modular multiplication is given in [6].

2 Multiplication by Shift-and-Add

It is worthwhile to recall the Shift-and-Add algorithm, since Reduce-by-Feedback is constructed completely analogously, retaining its properties:

Algorithm 1 *Shift-and-Add*
Parameters:
operand length l $[e.g. = 1024]$
shift length per clock cycle z $[e.g. = 3]$, *with* $Z := 2^z$ $[e.g. = 8]$
IN $A, B < 2^l$ // *factors, where* $A = \sum_{k=0}^{l-1} a_k 2^k = \sum_{k=0}^{\lceil l/z \rceil - 1} \alpha_k Z^{\lceil l/z \rceil - 1 - k}$
OUT M // *product* $M = A \cdot B$
Algorithm:
$M := 0$
`FOR` $k := 0$ `TO` $\lceil l/z \rceil - 1$
  $M := (M << z) + \alpha_k \cdot B$
`ENDFOR`

Some trivial, but remarkable properties of Shift-and-Add are:

(*i*) The coefficient α_k lies in the range $\{0, 1, \ldots, Z-1\}$, thus Z possible multiples of B are to be taken into account. Note that α_0 is the MSB part.

(*ii*) We have exactly $\lceil l/z \rceil$ cycles to go in the loop, a fixed timing.

(*iii*) It is sufficient to store the multiples for $\alpha \geq Z/2$, and $\alpha = 0$, by supplying shifted copies for the smaller cases, *e.g.* cases $3 \cdot B, 6 \cdot B$ (for $\alpha = 3$ and 6) from $12 \cdot B$, $\alpha = 12$ for $z = 4, Z = 16$.

(*iv*) The "1-off trick" [15, 16, 20, 7]: A further saving is possible by replacing the odd multiples by the next higher even ones, and subtract $Z \cdot B$ in the next clock cycle:
$((\alpha_k \cdot B) << z) + \alpha_{k+1} \cdot B = (((\alpha_k + 1) \cdot B) << z) + (\alpha_{k+1} - Z) \cdot B$.
Putting $C_{\alpha,k} := 1$, iff α_k is odd, 0 otherwise, we then set

$$\overline{\alpha}_k := \alpha_k + C_{\alpha,k} - Z \cdot C_{\alpha,k-1} \text{ and } M := (M << z) + \overline{\alpha}_k \cdot B.$$

Hence, (*iii*) and (*iv*) combined leave us with the necessary multiples $\pm(Z/2+2), \pm(Z/2+4), \ldots, \pm Z, 0$, where we first applied (*iv*), then (*iii*).

While these are still $Z/2$ choices, and including shifts we again have Z multiples, as are necessary by using base Z, the $\pm$ comes for free in hardware as two's complement, taking the inverse outputs $\overline{Q}$ of the register latches. Only the $Z/4$ multiples $Z/2+2, Z/2+4, \ldots, Z$ have to be stored in hardware.

3 Reduce-by-Feedback

History:
This algorithm was first introduced in 1987 by Vielhaber [15]*, also* [16]*, and in 1990 the German patent* [20] *was granted.*
Beth and Gollmann describe the algorithm in [2]*, in 1989.*
Benaloh and Dai rediscovered the algorithm and gave a talk at the Rump Session of CRYPTO'95 [1]*, patenting it in the United States in 1998 as* [19]*.*
Finally, Jeong and Burleson re-re-discovered the algorithm in 1997, when it appeared in the journal article [7]*.*

3.1 The Algorithm

The original idea stems from the analogy with LFSR's: The z bits running off in front for each Shift-and-Add step are fed back into the accumulator:

Let $K \equiv 2^{l+2z+1} \mod N, 0 \leq K < N$.

Also, partition M into its lower $l+z+1$ bits and the higher part, $M_H = \lfloor M/2^{l+z+1} \rfloor, M_L = M \mod 2^{l+z+1}$, $M = (M_H|M_L)$. Then

$$(M_H|M_L) << z = M_H \cdot 2^{l+2z+1} + M_L \cdot 2^z \equiv M_H \cdot K + M_L \cdot 2^z \mod N$$

The *Shift-and-Add-with-Reduce-by-Feedback* algorithm now runs as follows (note that μ_k is M_H):

Algorithm 2 *Shift-and-Add-with-Reduce-by-Feedback*
$M := 0, C_{\alpha,-1} := 0, C_{\mu,-1} := 0$
`FOR` $k := 0$ `TO` $\lceil l/z \rceil - 1$
  $C_{\alpha,k} := \alpha_k \mod 2, \overline{\alpha}_k := \alpha_k + C_{\alpha,k} - Z \cdot C_{\alpha,k-1}$
  $\mu_k := \lfloor M/2^{l+z+1} \rfloor$
  $C_{\mu,k} := \mu_k \mod 2, \overline{\mu}_k := \mu_k + C_{\mu,k} - Z \cdot C_{\mu,k-1}$
  $M := ((M \mod 2^{l+z+1}) << z) + \overline{\alpha}_k \cdot B + \overline{\mu}_k \cdot K$
`ENDFOR`
// $M = A \cdot B \mod N, 0 \leq M < 2^l$ (*not necessarily* $M < N$)

Reduce-by-Feedback preserves the 4 properties of Shift-and-Add:

(*i*) The standard range for the multiples of K is $\mu_k \in \{-1, 0, 1, \ldots, 2^z\}$.
(*ii*) The `FOR` loop excutes exactly $\lceil l/z \rceil$ times, each run comprising a shift and 2 additions. This amount is independent of the values.
(*iii*) The multiples of K required according to (*i*) can be restricted to $\mu_k \in \{0\} \cup \{Z/2+1, \ldots, Z\}$, supplying the others by shifting.
(*iv*) The odd multiples can be traded for negative ones, applying the "1-off trick". Hence in total we need $\alpha_k, \mu_k \in \{0, \pm(Z/2+2), \pm(Z/2+4), \ldots, \pm Z\}$, with 0 and $\pm$ for free in hardware.

Reduce-by-Feedback is thus *completely analogous* to Shift-and-Add.

3.2 Overflow Avoidance

We check overflow avoidance by proving the inequality

$$-Z \leq \overline{\mu}_k = M_H \leq Z, \forall k$$

by induction.

We have $0 \leq B, K < 2^l$ and $0 \leq M_L < 2^{l+z+1}$. Including the "1-off trick", we require $-Z \leq \overline{\alpha}_k, \overline{\mu}_k \leq Z$, and $\overline{\alpha}_k, \overline{\mu}_k$ being even. This is true for $\overline{\alpha}_k, \forall k$ and can be assumed for $\overline{\mu}_0 = 0$ at the start.

Then

$$-1 \cdot 2^{l+z+1} \leq (M_L << z) + \overline{\alpha}_k \cdot B + \overline{\mu}_k \cdot K < 2^{l+z+1} \cdot (2^z + 1/2 + 1/2)$$

i.e. $-1 \leq M_H^+ \leq 2^z$. As with C_α, we put $C_{\mu,k} = 1$, if μ_k is odd and has to be increased by the "1-off trick", $C_{\mu,k} = 0$ otherwise, and then have

$$\overline{\mu}_{k+1} = M_H^+ + C_{\mu,k+1} - C_{\mu,k} \cdot Z \in \{-Z, -Z+2, \ldots, Z-2, Z\},$$

which proves the induction step.

Therefore, the accumulator M never exceeds the range $-1 \cdot 2^{l+z+1} \leq M < (Z+1) \cdot 2^{l+z+1}$ and the even multiples of B up to $\pm Z \cdot B$ are sufficient.

4 Implementation Issues

4.1 Re-use of MUX Tree

Since the choice of the correct multiples, $\overline{\alpha}_k \cdot B + \overline{\mu}_k \cdot K$, is completely analogous for B and for K, we may use the same logic (calculation of decision variables, MUX tree, shifter) first for the part $\overline{\alpha}_k \cdot B$ (in one half clock cycle), and then for $\overline{\mu}_k \cdot K$ (in the other half clock cycle), as described in [15, 16, 20].

This 1:1 analogy between Shift-and-Add and Reduce-by-Feedback was the central idea of the algorithm and leads to very compact VLSI designs:

Mapping the implementation in [16] to current 65 nm rules, and naïvely assuming a shrinking factor $(65/1000)^2$, this would roughly lead to $13 \cdot (1000/65)^2 \approx 3000$ bits/mm^2, or a full 4096 bit RSA with control unit on about 1.5 square millimeters.

4.2 Delayed-Carry-Adder

Brickell [3] introduced the Delayed-Carry-Adder, a chain of halfadders instead of full adders, and where the resulting double register has the property $c_{i+1} \wedge s_i = 0$.

The advantage of the Delayed-Carry-Adder is the locality of carries. We do not have to wait for carry propagation and thus addition is fast. At the end of a multiplication, however, the final Delayed-Carry result has eventually to be added into the standard form, which may lead to a timing attack (see Section 5).

Nevertheless, without carry-save techniques, this carry propagation problem would arise at each addition intead of just once at the end.

Also, we have to take extra care when dealing with the upper part M_H (μ_k) of the accumulator, see next subsection.

The addition $(c,s)^+ := (c,s) + b + k$ usually requires two full adders in carry-save technique. With Brickell's delayed-carry scheme, we add as follows, where (c,s) is the delayed-carry register, (b) and (k) are the terms $\alpha \cdot B$ and $\mu \cdot K$, respectively. t, u, v are intermediate sum terms, d, e, f, g, h are intermediate carries. In NAND-logic, the variable c will only be used invertedly.

Standard Boolean function		Using NAND-2 gates	
$d_i := s_i \wedge b_i,$	$t_i := s_i \oplus b_i$	$\overline{d}_i := \overline{s_i \wedge b_i},$	$t_i := s_i \oplus b_i$
$e_i := t_i \wedge k_i,$	$u_i := t_i \oplus k_i$	$\overline{e}_i := \overline{t_i \wedge k_i},$	$u_i := t_i \oplus k_i$
$f_i := c_i \vee d_{i-1}$	(which are not both 1, due to $c_{i+1} \wedge s_i = 0$)	$f_i := \overline{\overline{c}_i \wedge \overline{d}_{i-1}}$	
$g_{i+1} := u_i \wedge f_i,$	$v_i := u_i \oplus f_i$	$\overline{g}_{i+1} := \overline{u_i \wedge f_i},$	$v_i := u_i \oplus f_i$
$h_{i+1} := e_i \vee g_i$	(not both 1: $e_i = 1 \Rightarrow u_i = 0$)	$h_{i+1} := \overline{\overline{e}_i \wedge \overline{g}_i}$	
$c^+_{i+1} := v_i \wedge h_i,$	$s^+_i := v_i \oplus h_i$	$\overline{c}^+_{i+1} := \overline{v_i \wedge h_i},$	$s^+_i := v_i \oplus h_i$

Table 1. Boolean logic for Delayed-Carry adder

This leaves us with 4 halfadders plus two OR's, the equivalent of two full adders. We thus need the same number of gate equivalents, but the result now has the Delayed-Carry Property $c_{i+1} \wedge s_i = 0$, which is crucial, when calculating μ_k (see next paragraph).

4.3 How to keep the invariant when using the delayed-carry representation

We feed back the z leading MSB bits, which have to be in the range $-1, 0, \ldots, Z$ (assumption for overflow avoidance).

With delayed-carry, we have $c_{i+1} \wedge d_i = 0$, hence the following patterns are the highest values possible (shown for the case $z = 3, Z = 8$), Table 1.

As can be seen in Table 2, cases 4 and 5 would lead to an overflow ($M_H > Z = 8$) due to the Delayed-Carry representation. We avoid this by looking further to the right and (cases 1 and 2) detect and avoid a subsequent overflow already in the previous cycle.

4.4 Fast computation of MUX control variables

It is crucial that the clock frequency depends only on the data propagation within the bit slices, and not on the control module.

In each clock cycle, we add $\overline{\alpha} \cdot B$ and $\overline{\mu} \cdot K$ to the delayed-carry register (c,s). In the two previous half cycles, we choose these multiples by the same

1	$c_{2^{l+z+1}+2,1,0;-1,-2}$	0	0 0	0 1	sum is 8 with carry, OK, avoids case 4
	$s_{2^{l+z+1}+2,1,0;-1,-2}$	1	1 1	1 1	
	$M_{H,2^{l+z+1}+3,2,1,0;-1,-2}$	1	0 0	0 0	
2	$c_{2^{l+z+1}+2,1,0;-1,-2}$	0	0 0	1 1	sum is 8 with carry, OK, avoids case 5
	$s_{2^{l+z+1}+2,1,0;-1,-2}$	1	1 1	1 0	
	$M_{H,2^{l+z+1}+3,2,1,0;-1,-2}$	1	0 0	0 1	
3	$c_{2^{l+z+1}+2,1,0;-1,-2}$	0	0 1	1 1	sum is 8, OK
	$s_{2^{l+z+1}+2,1,0;-1,-2}$	1	1 1	0 0	
	$M_{H,2^{l+z+1}+3,2,1,0;-1,-2}$	1	0 0	1 1	
4	$c_{2^{l+z+1}+2,1,0;-1,-2}$	0	1 1	1 1	sum is 9, to be avoided by case 1
	$s_{2^{l+z+1}+2,1,0;-1,-2}$	1	1 0	0 0	
	$M_{H,2^{l+z+1}+3,2,1,0;-1,-2}$	1	0 1	1 1	
5	$c_{2^{l+z+1}+2,1,0;-1,-2}$	1	1 1	1 1	sum is 11, to be avoided by case 2
	$s_{2^{l+z+1}+2,1,0;-1,-2}$	1	0 0	0 0	
	$M_{H,2^{l+z+1}+3,2,1,0;-1,-2}$	1	0 1	1 1	

Table 2. MSB sum of Delayed-Carry-Adder

hardware (MUX, shifter, logic), which is not time-critical for $\overline{\alpha} \cdot B$, since in principle, all values α are known. On the other hand, $\overline{\mu}$ depends on the addition just performing in the half cycle $(k+1, H)$, while the next multiple $\overline{\mu} \cdot K$ must be selected in $(k+1, L)$. We proceed as follows (see [15][16]):

Clock cycle	Half cycle	Selection	Computation
k	H	$\overline{\alpha}_k \cdot B$	$(M_H \vert M_L)_k := \ldots$
k	L	$\overline{\mu}_k \cdot K$	
$k+1$	H	$\overline{\alpha}_{k+1} \cdot B$	$(M_H \vert M_L)_{k+1} := ((M_L)_k << z) + \overline{\alpha}_k \cdot B + \overline{\mu}_k \cdot K$
$k+1$	L	$\overline{\mu}_{k+1} \cdot K$	

Table 3. Precomputation of control variables

Having calculated $(M_H)_{k+1}$ in half cycle $(k+1, H)$, immediately afterwards we need $\overline{\mu}_{k+1}$ in half cycle $(k+1, L)$. We therefore have to precompute as much as we can: In (k, H), we already compute a partial sum $(M_H)_k \cdot Z + \overline{\alpha}_k \cdot B$ for the bit positions of M_H, including 2 more bits to the right, as described in the previous paragraph, to avoid possible overflow in the future. We then add the part $\overline{\mu}_k \cdot K$ in (k, L), for these bit positions. We also add 0,1,2,3 to obtain the four possible final values for $\overline{\mu}$, and for all four possibilities, we precompute the MUX control variables for the next choice of $\overline{\mu} \cdot K$. The only missing part are up to 3 carries from the lower part, M_L, of the sum. In this way, terminating $(k+1, H)$, we obtain the new sum $(M_H)_{k+1}$, and immediately select the MUX-control values to fetch $\overline{\mu}_{k+1} \cdot K$ in $(k+1, L)$ from the 4 precomputed sets.

The full-custom implementation in [16] achieves a control unit faster than the bit slices. We have this design goal also for the FPGA implementation. It remains to be verified though, whether this will apply or whether the FPGA architecture (6-input LUTs instead of a chain of half-adders) will actually make the bit slices even faster.

5 Timing Attacks

We may trivially find the Hamming weight of the exponent by just counting multiplications and squarings. To prevent this, we would have to either do both in parallel, wasting space, or introduce dummy multiplications, wasting time.

In any case, this issue is independent of the implementation of modular multiplication.

As Kocher [9] points out, however, apart from the Hamming weight, we can indeed recover the full exponent — provided that multiplication time is sensitive on the values, some lead to faster calculation than others.

The attack by Schindler [13] on Montgomery multiplication can easily be overcome by introducing a dummy subtraction, costing a single clock cycle. There is no analogue of this attack against Reduce-by-Feedback.

Therefore, with Reduce-by-Feedback as well as with Montgomery multiplication (+dummy), timing attacks are ruled out during the modular multiplication, taking in any case exactly $\lceil l/z \rceil$ cycles. The result is then in a delayed-carry- or carry-save-register.

The final carry however, may introduce timing information. Either

(i) we use carry-look-ahead logic, space-intensive, or
(ii) we keep the result in delayed-carry-form, space-intensive, or
(iii) we wait until the longest carry chain ($l + z$ bits) will have passed, time-intensive, or
(iv) we use interrupt techniques, efficient, but time-variant.

The variation due to carries in case (iv) is the only potential information leak for a timing attack. This is though independent of Reduce-by-Feedback (or Montgomery multiplication), but a consequence of using carry-save or delayed-carry techniques.

Up to here, this concerned the modular multiplication as building block. As to the exterior loop, exponentiation, Square-and-Multiply, there must of course be the same number of clock cycles between any two multiplications and/or squarings to avoid a timing/DPA mix just concentrating on the transition between two of them. Otherwise, use the double-add scheme by Joye [8] in the multiplicative version "square-multiply", wasting time though. However, this does not concern modular multiplication proper, but exponentiation.

6 How to break RSA with Differential Power Analysis

Both Reduce-by-Feedback and Montgomery multiplication make RSA susceptible to the following DPA [10] attack. For other attacks against RSA see the power attack by Yen *et al.* [18], and the timing attack by Miyamoto *et al.* [11].

Now to our DPA attack: Every multiplication (in this section this includes squarings) starts with an empty accumulator $M = 0$, and also a zero adjustment value μ_0 (both for Reduce-by-Feedback and Montgomery multiplication).

The first factor, A, will on average start with z zeroes every Z'th multiplication. In this case, $\alpha_0 \cdot B = 0$, while the term will be nonzero otherwise.

For $\mu_0 = \alpha_0 = 0$ (in terms of Reduce-by-Feedback), we compute

$$M^+ = (M << z) + \alpha_0 \cdot B + \mu_0 \cdot K = (0 << z) + 0 + 0 = 0,$$

hence the register M was empty before the step and is overwritten again with zeroes.

If, on the other hand, $\alpha_0 \neq 0$,

$$M^+ = (M << z) + \alpha_0 \cdot B + \mu_0 \cdot K = 0 + \alpha_0 \cdot B + 0 \neq 0,$$

and roughly half of the flip-flops of register M will change state from 0 to 1. This gives a strong difference in power consumption during this first cycle of the multiplication, compared to $M^+ = M = 0$, a "point-of-interest" in terms of the template attack [4].

We focus only on this information (about half a bit for $z = 3$) and will assume that we can distinguish between $A < \frac{1}{Z} \cdot 2^l$, case $\alpha_0 = 0$, and $A \geq \frac{1}{Z} \cdot 2^l$, case $\alpha_0 \neq 0$, for every multiplication step.

We assume that we have access to the public RSA modulus N and to several known ciphertexts $\chi_1, \chi_2, \ldots$. We observe the decryptions $\chi_i^d \mod N$ for a fixed unknown exponent d (unblinded case). We compute the multiplication chains for all 2^L possible initial segments of d of a certain length L. These segments will consist of L squarings and furthermore L', $0 \leq L' \leq L$, multiplications, depending on the number of 1's in the segment. For each hypothetical segment, we do the corresponding calculations (multiplications and squarings) and memorize the sequence of initial coefficients α_0 of length $L + L'$.

We now observe the actual H/W decryption and obtain a sequence $\{= 0, \neq 0\}^{2L}$, whose first $L+L'$ components we check against all possible initial segments.

The per-symbol information is $-\left(\log_2\left(\frac{1}{Z}\right) \cdot \frac{1}{Z} + \log_2\left(\frac{Z-1}{Z}\right) \cdot \frac{Z-1}{Z}\right) = 1.0$, 0.811, 0.544, and 0.337 bits for $z = 1, 2, 3$, and 4, respectively. Hence, 1,2,2,3 decryptions χ_i should be sufficient.

The crucial case is, however, the large set of initial segments leading to the sequence $(\neq 0)^{L'}$, in the case that this is the actual observation. We expect this to happen with probability $\left(\frac{Z-1}{Z}\right)^{L'}$, thus leading to $\left(\frac{Z-1}{Z}\right)^{L'} \times 2^L$ cases. We set $L' := 0.5L$ from now on and consider C decryptions $\chi_1, \ldots, \chi_C$, whose outcomes ($\alpha = 0$ or $\alpha \neq 0$) we assume independent.

The expected number of segments which always lead to $(\neq 0)^{L+L'}$, in all C decryptions, is then

$$\left(\frac{Z-1}{Z}\right)^{1.5L\cdot C} \times 2^L.$$

To have uniqueness, we want this size down to 1, hence $\left(\frac{Z-1}{Z}\right)^{1.5L\cdot C} \times 2^L = 1$ or $C = -1/(1.5\log_2(7/8))$, which gives $C = 0.67, 1.61, 3.47$, and 7.16 for $z = 1, 2, 3$, and 4, respectively. Therefore, $C \geq Z/2$ samples (asymptotically $Z \cdot \ln(2)/1.5$ samples) are necessary.

We now compare the $C \geq Z/2$ sequences actually observed from $\{= 0, \neq 0\}^{L'}$ with all initial segments of d, saving only the matches, where under ideal conditions, only a single match should occur. These matches are then extended, compared to the observations, and so forth, until recovering the full secret RSA exponent d.

Certainly, there will be noise in our measurements, so quite some more than $Z/2$ ciphertexts will be needed under realistic conditions.

And that breaks RSA!

7 How to repair Reduce-by-Feedback to avoid the DPA attack on RSA

In this section, we suggest modifications to strengthen Reduce-by-Feedback against Differential Power Analysis.

As we have seen, the initial all-zero phase is exploitable by DPA. We can neither avoid $\mu_0 = 0$ in the first step, nor $\alpha_0 = 0$ once in a while — if using directly the z bits of A, and M_H, respectively.

We can, however, avoid $M = 0 \mapsto M^+ = 0$ in this cases, by using the same "1-off" trick as in property (iv) of Shift-and-add and Reduce-by-Feedback:

$$0 = 1 + (-1)$$

We just never add a zeroth multiple, but instead add B once, and subtract it (Z-fold) in the next step. This brings us back to zero every second step. Assuming B to have 50% 1's, the effect is flipping back-and-forth half of the register bits.

To be explicit, we use the case $z = 3, Z = 8$ in the sequel. The columns "old" show the regular case [15, 16, 20], applying properties (iii) and (iv), including a multiple 0. We also adjust the treatment of values $\Sigma = -1, 1, 2$, and 3 to minimize the information flow (bias) from $\overline{\alpha}, \overline{\mu}$ to C, A, M_H, see columns "new".

Note that we still use the "1-off trick", however in an irregular way, so that the required multiples are no longer just the even ones. In any case, all required multiples can still be obtained by shifting from only $Z/4$ values, *e.g.* 6, and 8.

Description of Table 4, multiples $\overline{\alpha}_k, \overline{\mu}_k$ from A, M_H

The original α_k (bits from A), may vary from 0 to $Z - 1 = 7$, M_H (upper part of M) may vary from -1 to $Z = 8$. Applying property (iv), a previous odd

C_α, C_μ	α_k, M_H	Σ	$\overline{\alpha}_k$, $\overline{\mu}_k$(old)	C^+	$\overline{\alpha}_k$, $\overline{\mu}_k$(new)	C^+
0	−1	−1	0	1	−1	0
0	000	0	0	0	1	1
0	001	1	2	1	1	0
0	010	2	2	0	3	1
0	011	3	4	1	3	0
0	100	4	4	0	4	0
0	101	5	6	1	6	1
0	110	6	6	0	6	0
0	111	7	8	1	8	1
0	1000	8	8	0	8	0

C_α, C_μ	α_k, M_H	Σ	$\overline{\alpha}_k$, $\overline{\mu}_k$(old)	C^+	$\overline{\alpha}_k$, $\overline{\mu}_k$(new)	C^+
1	−1	−9	−8	1	−8	1
1	000	−8	−8	0	−8	0
1	001	−7	−6	1	−6	1
1	010	−6	−6	0	−6	0
1	011	−5	−4	1	−4	1
1	100	−4	−4	0	−3	1
1	101	−3	−2	1	−3	0
1	110	−2	−2	0	−1	1
1	111	−1	0	1	−1	0
1	1000	0	0	0	1	1

Table 4. Old and new multiples $\overline{\alpha}_k, \overline{\mu}_k$

value was adjusted by +1, hence we may have to adjust now ($C_\alpha, C_\mu = 1$) by $-Z = -8$, giving an overall sum Σ between −9 and +8. Σ is now split into a multiple actually added, $\overline{\alpha}_k, \overline{\mu}_k$, minus a possible new carry $C^+_\alpha, C^+_\mu = 1$. In the original scheme, the multiples were $0, \pm 2, \pm 4, \pm 6$, and ± 8, while we now have $\pm 1, \pm 3, \pm 4, \pm 6$, and ± 8, avoiding zero.

Observe that in both cases, all multiples are shifts and negatives of just the two multiples 6 and 8. Hence, even after the modification, only these 2 multiples have actually to be stored (and computed).

Description of Table 5, Bias

There is now less bias between Σ, C and the bits of α_k, μ_k. We define bias as $\mathrm{pr}(1) - \mathrm{pr}(0)$ (not as $\mathrm{pr}(1) - \frac{1}{2}$).

We assume probability 1/8 each for $\alpha = 0, \ldots, 7$. For, μ, by folding 3 equidistributions over the intervals $[0, 8[$, $[-1/2, 1/2[$, and $[-1/2, 1/2[$, we obtain probability 1/8 each for $\mu = 1, \ldots, 6$, probability 5/48 for $\mu = 0$ and 7, and probability 1/48 for $\mu = -1$ and 8, each comprising the interval $M_H \in [\mu, \mu + 1[$.

$C = 0$ and $C = 1$ are each assigned probability 1/2.

We consider the bias of the bits of C and Σ (internal values revealing information about the actual contents of A and M), conditional on certain value sets for $\overline{\alpha}, \overline{\mu}$, namely zero, positive, shifts of 8, and shifts of 6 (potentially observable by DPA).

We now have probability zero for $\overline{\alpha} = 0$, which was 1/8 before. Neither can we infer anything on observing a shift of 8 (1,2,4,8) vs. a shift of 6 (3,6).

What remains is a bias from $\overline{\alpha}$ positive to $C = 0$ (which is almost a tautology). The fact $\overline{\alpha} > 0$, however, is a mix of the cases $\overline{\alpha} = 1, 2, 3, 4, 6, 8$, far more difficult to analyze by DPA than the distinction $\alpha = 0$ vs. $\alpha \neq 0$, now ruled out.

We now give the complete Shift-and-Add-with-Reduce-by-Feedback algorithm for $z = 3$, including the mentioned modifications, and the final adjustment from delayed-carry to a single register.

	$C\|\alpha$	$\Sigma_2\|\alpha$	$\Sigma_1\|\alpha$	$\Sigma_0\|\alpha$	$C\|\mu$	$\Sigma_2\|\mu$	$\Sigma_1\|\mu$	$\Sigma_0\|\mu$
$\overline{\alpha},\overline{\mu}=0$ new=old	0	0	0	0	0	0	0	0
$\overline{\alpha},\overline{\mu}>0$ new	-1	0	0	0	$-23/24$	$1/24$	$1/24$	$1/24$
$\overline{\alpha},\overline{\mu}>0$ old	-1	$1/7$	$1/7$	$1/7$	-1	$-2/21$	$-2/21$	$-2/21$
$\overline{\alpha},\overline{\mu}\in\{\pm1,\pm2,\pm4,\pm8\}$ new=old	0	0	0	0	0	0	0	0
$\overline{\alpha},\overline{\mu}\in\{\pm3,\pm6\}$ new=old	0	0	0	0	0	0	0	0

Table 5. Bias of C,Σ, conditional on $\overline{\alpha},\overline{\mu}$

Algorithm 3 *Shift-and-Add-with-Reduce-by-Feedback*
IN A,B,N // each at most l bits long, N odd
OUT M // the product $M = A\cdot B \mod N, 0 \le M < 2^l$ (not necessarily $M<N$)
// M *is actually stored in a delayed-carry register* (c,s). *Table* 2 :
`const mult[-9..8]` $= (-8,-8,-6,-6,-4,-3,-3,-1,-1,1,1,3,3,4,6,6,8,8)$
`const C[-9..8]` $= (1,0,1,0,1,1,0,1,0,1,0,1,0,0,1,0,1,0)$
$M:=0, C_\alpha:=0, C_\mu:=0$
`FOR` $k:=0$ `TO` $\lceil l/z\rceil - 1$
  $\alpha := 4\cdot a_{3k+2} + 2\cdot a_{3k+1} + 1\cdot a_{3k} - 8\cdot C_\alpha$
  $\overline{\alpha} :=$ `mult`$[\alpha], C_\alpha :=$ `C`$[\alpha]$
  $\mu := \lfloor M/2^{l+z+1}\rfloor - 8\cdot C_\mu$
  $\overline{\mu} :=$ `mult`$[\mu], C_\mu :=$ `C`$[\mu]$
  $M := ((M \mod 2^{l+z+1}) << z) + \overline{\alpha}\cdot B + \overline{\mu}\cdot K$
`ENDFOR`

// *Multiply by* 2^9
`FOR` $k:=1$ `TO 3`
  $\overline{\alpha} := -8\cdot C_\alpha, C_\alpha := 0$
  $\mu := \lfloor M/2^{l+z+1}\rfloor - 8\cdot C_\mu$
  $\overline{\mu} :=$ `mult`$[\mu], C_\mu :=$ `C`$[\mu]$
  $M := ((M \mod 2^{l+z+1}) << z) + \overline{\alpha}\cdot B + \overline{\mu}\cdot K$
`ENDFOR`
// *Divide by* 2^9, *leaving* $M < 2^l$
`FOR` $k:=1$ `TO 9`
  `IF` M `is odd` $N' := N$ `else` $N' := 0$
  $M := (M + N') >> 1$
`ENDFOR`
$M := C + S$ // *the final carry, using e.g. carry-look-ahead or interrupts*

Although N' is either N or zero in the last 9 steps, the result $(M+N') >> 1$ will differ from M in about half of the bits in both cases, making DPA based on flip-flop recharges extremely difficult.

Unfortunately (or luckily, if we want to promote Reduce-by-Feedback), we see no way to implement this modification with Montgomery multiplication:

The two properties (*iii*) and (*iv*) of Shift-and-Add-with-Reduce-by-Feedback can be mapped to Montgomery multiplication as

(*iii*) use shifted multiples (of N) to compensate results terminating in $\ldots 0$, and (*iv*) use the 2's complement of multiples of N terminating in $\ldots 01$ to account for those terminating in $\ldots 11$.

Again, we have a total of $Z/4$ multiples physically to be stored, those multiples of N terminating in $\ldots 01$. However, there seems to be no workaround to replace the do-nothing (subtract $0 \cdot N$) in the case $\ldots 000$ by anything else.

Conclusion

We have (re-)introduced the Reduce-by-Feedback algorithm, which can be seen as "Montgomery on the high end", but was inspired by LFSR feedback.

Reduce-by-Feedback is immune against timing attacks (as is Montgomery multiplication with dummy subtraction), with the possible exception of the final carry run.

We recalled how to avoid physically storing multiples, by providing shifted multiples, and using the "1-off trick", saving 75%.

RSA can be broken by DPA, when executed with Montgomery multiplication, or the unmodified Reduce-by-Feedback.

We proposed modifications for the choice of multiples of both the second factor B and the feedback value $K \equiv 2^{l+2z+1} \mod N$. These modifications diminish bias, avoid the multiple zero, and thereby avoid the accumulator being zero in consecutive time steps. These effects of the modification will diminish the susceptibility of Reduce-by-Feedback to Differential Power Analysis considerably. In particular, the DPA attack of Section 6 on RSA, exploiting the partial multiplier zero, is no longer possible.

Replacing a multiple zero with "$1 + (-1)$" by the "1-off trick" is not possible for Montgomery multiplication. Therefore, the DPA attack against RSA with Montgomery multiplication is still possible.

We have therefore shown that *Reduce-by-Feedback-with-Shift-and-Add* is the method of choice, to implement a timing-resistant and DPA-aware modular multiplication.

References

1. Josh Benaloh and Wei Dai, *Fast Modular Reduction*, CRYPTO'95 Rump Session.
2. Thomas Beth, Dieter Gollmann, *Algorithm engineering for public key algorithms*, IEEE J. SAC **7(4)**, 458–466, 1989.
3. Ernest F. Brickell, *A Fast Modular Multiplication Algorithm with Applications to Two Key Cryptography*, Proc. CRYPTO 82, 51–60, Plenum Press, 1983.
4. Suresh Chari, Josyula R. Rao, Pankaj Rohatgi, *TemplateAttacks*, Proc. Cryptographic Hardware and Embedded Systems CHES 2002, LNCS 2523, 51–62, Springer, 2003.
5. Adam J. Elbirt, Christof Paar, *Towards an FPGA architecture optimized for public-key algorithms*, The SPIE's Symposium on Voice, Video, and Communications, Boston, 1999.

6. Jorge Guajardo, Sandeep S. Kumar, Christof Paar, Jan Pelzl, *Efficient Software-Implementation of Finite Fields with Applications to Cryptography*, Acta Appl. Math **39**, 75–118, 2006.
7. Yong-Jin Jeong, Wayne P. Burleson, *VLSI array algorithms and architectures for RSA modular multiplication*, IEEE Trans. VLSI Systems **(5)2**, p. 211–217, 1997.
8. Marc Joye, *Highly regular right-to-left algorithms for scalar multiplication*, Proc. Cryptographic Hardware and Embedded Systems CHES 2007, LNCS 4727, 135–147, Springer, 2007.
9. Paul C. Kocher, *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*, CRYPTO '96, LNCS 1109, 104–113, Springer, 1996.
10. Paul C. Kocher, Joshua Jaffe, Benjamin Jun, *Differential Power Analysis* CRYPTO '99, LNCS 1666, 388–397 , Springer, 1999.
11. Atsushi Miyamoto, Naofumi Homma, Takafumi Aoki, Akashi Satoh, *Enhanced Power Analysis Attack Using Chosen Message against RSA Hardware Implementations*, ISCAS 2008, Seattle, 3282–3285.
12. Peter L. Montgomery, *Modular Multiplication without trial Division*, Math. Comp. **44**, 519–521, 1985.
13. Werner Schindler, *A timing attack against RSA with the Chinese remainder theorem*, Proc. CHES 2000, pp.109–124, Springer, 2000.
14. Holger Sedlak, Uwe Golze, *An RSA Cryptography processor*, Proc. Euromicro '86, Microprocessing and Microprogramming **18**, 583–590, 1986.
15. Michael Vielhaber, *Entwurf und Layout eines RSA-Koprozessors für Chipkarten*, Diploma Thesis, TH Karlsruhe (KIT), 1987.
16. Michael Vielhaber, *The Karlsruhe RSA Co-processor: ISDN Network Security by RSA encryption*, E.I.S.S. Report 89/14a, European Institute for System Security, Karlsruhe, 1990.
17. Michael Vielhaber, *Der Karlsruher RSA Koprozessor: Verschlüsseln mit RSA im ISDN-Netz*, E.I.S.S. Report 89/14, European Institute for System Security, Karlsruhe, 1990.
18. Sung-Ming Yen, Wei-Chih Lien, SangJae Moon, JaeCheol Ha, *Power Analysis by Exploiting Chosen Message and Internal Collisions Vulnerability of Checking Mechanism for RSA-Decryption*, Proc. MyCrypt 2005, LNCS 3715, 192–204, 2005.
19. USPTO Patent US5724279: *Computer-implemented method and computer for performing modular reduction*, Applicants: Josh Benaloh, Wei Dai.
20. Deutsches Patentamt, DE P 3924344 *Multiplikations-/Reduktionseinrichtung*, 1992, Anmelder: Vielhaber, Michael Johannes.