THE PROOF BY 2M-1: A LOW-COST METHOD TO CHECK ARITHMETIC COMPUTATIONS

Sylvain Guilley and Philippe Hoogvorst
Ecole Nationale des Télécommunications, 46, rue Barrault, 75634 Paris CEDEX 13, France.
E-mail :sylvain.guilley@enst.fr ,philippe.hoogvorst@enst.fr

Abstract: Injecting faults into an arithmetic device is a way of attacking cryptographic devices.The *proof by 2^m-1* is a method to detect arithmetic errors induced by this attack without having to duplicate the computations. This method is simple and not too expensive, in terms of computation power when the arithmetic in software and in terms of both silicon surface and power consumption when the arithmetic operations are performed by a hard-wired operator. In that the *proof by 2^m-1* is well-suited for martcards, in which these resources are limited. The *proof by 2^m-1* is scalable, in that the designer can choose the parameter m, which determines the level of protection offered and the resources needed for the verification.

Key words: Fault injection, modular computations, Security, Modular Computations.

1. INTRODUCTION

The injection of errors during a cryptographic computation is an efficient attack provided the attacker has access to the target. A smartcard is its typical target. Its power is such that a single uccessful fault induction is enough to break RSA cite [Boneh et *al., 1997, Siefert, 2002]*.

The *proof by 2^m-1* is a self-test method which can verify arithmetic computations either in Z or in Z/nZ, for some (big) integer n and thus detect such fault induction. It is specific to countering fault injection: it does not address any other class of attack, such as DPA [Kocher et *al.,* 1999], SPA, timing attack [Kocher et *al.,* 1996] or EMA [Quisquater and Samyde, 2001].

The *proof by 2^m-1* is scalable, in that the designer can choose the parameter m, which determines the level of protection offered and the resources needed for the verification.

All cryptographic applications based on arithmetic operations can take advantage of the *proof by 2^m-1*, such as, [Rivest, Shamir and Adleman, 1978], the Diffie & Hellman protocol [Diffie and Hellman, 1976].

Section 2 exposes the principles of using arithmetic residue codes to check arbitrary size arithmetic computations. Section 3 shows the case of decimal computation.Then we switch to a larger modulus. Section 4 exposes how to check modular multiplications. Section 5 concludes the article.

2. PRINCIPLES OF THE *PROOF BY 2^M-1*

Let $(Z,+,\times)$ be the ring of integers, $M>1$ an integer, $(Z/MZ,\hat{+},\hat{\times})$ the ring of residual classes modulo M, and $P_M(.)$ be the canonical ring homomorphism $(Z,+,\times)\rightarrow(Z/MZ,\hat{+},\hat{\times})$.

The *proof by 2^m-1* uses the properties of $P_M(.)$ to verify the computations performed by a cryptographic device subject to fault injection with a small computational overhead, which is nearly independent of the size of the operands.

Other works [Noufal and Nicolaidis, 1999] use similar concepts but focuses on the synthesis of hardware multipliers. The authors use dedicated structures for fixed-width data paths that cannot be scaled to an arbitrary operand size.

Our solution can be implemented using either regular software or off-shelf hardware structures, available in CAD vendors libraries.

3. VERIFYING COMPUTATIONS IN Z

3.1 Proof by 9

When no pocket calculator is available, a simple way exists to check multiplications, which is a lot simpler than doing them twice: the *proof by 9*, which consists in repeating the multiplication modulo 9. If the actual result of the multiplication and the one obtained with the reduced operands are not congruent modulo 9, one of the results is wrong.

Calculating in $Z/9Z$ is easy because any operation involves only 1-digit numbers. As for any positive integer i, 10^i is congruent to 1 modulo 9, reducing a number n modulo 9 comes down to adding its decimal digits and reiterating the process until the sum is strictly smaller than 10. At the end a result equal to 9 is replaced by a 0 if necessary.

This process is guaranteed to stop because, if n consists $p > 1$ digits in base 10, i.e. $n = \sum_{0}^{p-1} n_i 10^i$, then $n' = \sum_{0}^{p-1} n_i$ is strictly smaller than n. The successive sums form a strictly decreasing sequence of integers, which must fall below 10 after a finite number of steps.

The process of reduction is roughly linear in complexity with the size of the operands but the complexity of the multiplication in $Z/9Z$ is independent of this size as it always consist of always a single 1-digit by 1-digit multiplication, followed by the reduction of a 2-digit number.

The *proof by 9* **does not prove correctness**. To prove correctness the process should be repeated with different prime numbers until the chinese remainder theorem could be used. However $P_9(x)P_9(y) \neq P_9(xy)$ proves that an error occurred. Otherwise a random mistake is detected with probability 1/9. In addition, the *proof by 9* will detect any error which affects a single digit, except if a 9 was replaced by a 0 or vice versa.

3.2 Proof by M-1

A 1/9 probability of missing an error is not safe enough for security purposes. Besides, if the operation $A \times B$ was to be checked, computing $P_9(A)$ and $P_9(B)$ would involve a lot of computations, each of them possibly subject to other attacks.

However changing the modulus will at the same time yield a much better fault coverage and make it very easy to compute the proof. In particular, any error affecting a single digit in base M will be detected except if a $M-1$ is replaced by a zero or vice versa. As no single-bit fault can do that, any single-bit fault will always be detected. Again the cost of the computation modulo $M-1$ will be a lot lower than the one of the real computation. The *proof by* $M-1$ is thus a lot cheaper than the repetition of the computations.

3.3 Notations

Let $M>1$ an integer. Though M will usually be 2^m, with $m \in \{8,16,32,64\}$, we will not use this fact in the rest of the article more than deriving the name of the method from it.

An uppercase letter X represents a positive integer, whose base B decomposition is denoted $\left(\ldots, X^{[2(B)]}, X^{[1(B)]}, X^{[0(B)]}\right)$, in which the $X^{[i(B)]}$ are an infinite sequence of integers in $[0, B-1]$ among which only a finite number of them is non-zero. Given a positive integer X, its size in base B, denoted $|X|_B$, is the smallest integer x such that $\forall j \geq x, X^{[j(B)]} = 0$.

To simplify the notations from now on we will use some shorthands.
Unless otherwise specified:

- the canonical ring homomorphism $Z \rightarrow Z/(M-1)Z$ is denoted $P(.)$ instead of $P_{M-1}(.)$,
- $\hat{+}$ and $\hat{\times}$ operate in $Z/(M-1)Z$,
- the digits of X in base M are denoted X^i instead of $X^{(i[M])}$,
- the size of X in base M is denoted $|X|$ instead of $|X|_M$.

3.4 Mathematical basis

It is easy to prove that, for any positive integer n and any number M,

$$\forall n > 0, (M^n - 1) = (M-1)(M^{n-1} + M^{n-2} + ... + M + 1).$$

Computing modulo $M-1$, this identity becomes:

$$\forall n > 0, M^n \equiv 1 \bmod (M-1) \quad (1)$$

From Eq. (1), given $A = \sum_{i=0}^{|A|} A^{[i]} M^i$, the evaluation of $A \bmod (M-1)$ consists in the addition of the $A^{[i]}$ with the following peculiarities:

1. a $\text{Carry}_{\text{out}}$ has weight M, thus is congruent to 1 modulo $(M-1)$, it can be re-injected as the Carry_{in} of the next addition;
2. the reduction ends with the addition of a zero to ensure that the last $\text{Carry}_{\text{out}}$ is effectively added. This last addition can produce no $\text{Carry}_{\text{out}}$ because, at the preceding addition, each of the operands was at most $(M-1)$.Thus the value of the result was at most $2M-2$, which can be rewritten as $M+(M-1)-1$ and, given than $M \equiv 1 \bmod (M-1)$, the result of the last addition will be at most $(M-1)$ and no $\text{Carry}_{\text{out}}$ can be generated;
3. at the end if the result is exactly $(M-1)$, it is replaced by zero.

The number of additions needed to reduce a number A modulo $(M-1)$ is $|A|+1$. It is proportional to $\log A$ and inversely proportional to $\log M$.

If this reduction has to be implemented in hardware, the data path consists in a conventional adder, in which the $\text{Carry}_{\text{out}}$ is connected to the Carry_{in}, together with the control circuitery which instructs the device to perform the final addition of a zero and, if necessary, the replacement of a $(M-1)$ by 0.

3.5 Checking additions and multiplications in Z

When computing with integers, the verification process is straightforward: given two operands A and B, an operation $*$ (which can be any of $+$, $-$ or $\times$) is performed and verified as follows:

1. evaluate $a = P(A)$ and $b = P(B)$,
2. evaluate $R = A * B$ in Z,
3. evaluate $r = a \hat{*} b$ in $Z/(M-1)Z$,
4. check that $P(R) = r$.

The operations involved are: the evaluations of $P(A)$ and $P(B)$, which cost $|A|+|B|+2$ additions, the evaluation of $P(R)$, which costs $|R|+1$ additions, the $\hat{*}$ operation between a and b, which is a single addition or multiplication, the reduction modulo $(M-1)$ of the result, which costs two additions at most.

Although they add overhead, the reductions of the operands must be performed only once, for the original operands. During the computation the reduced results of any operation will be kept together with the real results for use in the next operations. In that the overhead will be nearly independent of the computation performed.

3.6 Security of the test

Again the *proof by 2^m-1* can only prove non-correctness: that the results of the integer and of the modular operations are not congruent proves that an error occurred. Otherwise, the probability of an undetected error is exponentially decreasing with $|M|_2$ as shown on the table below.

1. $\|M\|_2$	1. Probability of an undetected error
2. 8	2. $2^{-8} = 3.92\times10^{-3}$
3. 16	3. $2^{-16} = 31.53\times10^{-5}$
4. 32	4. $2^{-32} = 2.33\times10^{-10}$
5. 64	5. $2^{64} = 5.42\times10^{-20}$

4. CHECKING MODULAR COMPUTATIONS

Cryptographic operations seldom involve multiprecision computations in Z itself. Most of the time, the computations are done in Z/nZ, for some integer number n. As there exists no non-trivial ring homomorphism from Z/nZ into $Z/(M-1)Z$ (unless n is a multiple of $(M-1)$) the checking of the computations is a little more difficult.

4.1 Modular Addition

Given two operands in Z, A and B, each of them being in $[0, N-1]$, their addition modulo N is performed in three steps:

1. compute $R'= A+B$ in Z. As $A<N$ and $B<N$, $R'<2N$;
2. if $R'<N$ set $R=R'$ else set $R=R'-N$.

None of these low-level operations is modular. It is thus possible to check the modular addition with two operations in $Z/(M-1)Z$:

1. compute $r'= P(A)\hat{+}P(B)$;
2. if $R'<N$ set $r=r'$ else set $r=r'-P(N)$.

Normally $P(N)$ will have been precomputed when N was set. Verifying a modular addition costs thus only one addition more than verifying it in Z.

4.2 Modular Multiplication

The multiplication modulo N of A and B, both in $[0,N-1]$ can be performed as:

1. let $T=A\times B$,
2. let Q and R be respectively the quotient and the remainder of the division of T by N,
3. the result is R.

As $A\times B=Q\times N+R$, the relation $P(A)\hat{\times}P(B)=P(Q)\hat{\times}P(N)\hat{+}P(R)$ must hold, which allows us to check the modular multiplication. However, a modular multiplication is never computed like that: the division is too expensive.

4.3 Binary modular multiplication

When no hard-wired multiplier is available, the multiplication in Z consists in a sequence of additions, together with left shifts for the multiplicand and bit tests for the multiplicator. Note that in all figures, the notation (X,x) stands for the couple $(X,P(X))$.

To transform the binary integer multiplication into a verifiable modular multiplication, we will first pretend to compute in Z but in base N. Each of B and P will be represented as digits in base N. As we are computing modulo N, we assume that $A<N$ and $B<N$, thus we have: $P=A\times B<N^2$ and $|P|_N\leq 2$. As for B, its value will be under N^2 until the beginning of the last round of computation. The last doubling may yield a wrong result but we don't use this last B. Fig. 4.3.1 shows the multiplication with base N computations. For the sake of readibility, we

have replaced $P^{[1(N)]}$ by P_h, $P^{[0(N)]}$ by P_l, $B^{[1(M)]}$ by B_h and $B^{[0(M)]}$ by B_l.

1. mulmod$\big((A,a),(B,b),(N,n)\big)$	1.
1. {	2.
1. $P_h \leftarrow 0; P_l \leftarrow 0; B_h \leftarrow 0; B_l \leftarrow 0;$	3. // Initialize
1. **for**($i \leftarrow 0$; $i < \lvert A \rvert_2$; $B_{li} \leftarrow 0$ i+1) {	4. // Loop on bits of A
1. **if**($A^{[i(2)]} \neq 0$) {	5. // Addition necessary?
1. $P_h \leftarrow P_h + B_h; P_l \leftarrow P_l + B_l;$	6.
1. **if**($P_l \geq N$){ $P_h \leftarrow P_h + 1; P_l \leftarrow P_l - N;$}	7. // Correction modulo N
1. }	8.
1. $B_h \leftarrow B_h + B_h; B_l \leftarrow B_l + B_l;$	9. // Double B
1. **if**($B_l \geq N$){ $B_h \leftarrow B_h + 1; B_l \leftarrow B_l - N;$}	10.// Correction modulo N
1. }	11.
1. Check that $a \hat{\times} b \equiv P(P_h) \hat{\times} P(N) \hat{+} P(P_l)$;	12.// Final check
1. **return** $\big(P_l, P(P_l)\big)$;	13.// Return both real and
2. }	14.// reduced results.

Fig. 4.3.1 : binary integer multiplication in base N.

As only the projection of P_h into $Z/(M-1)Z$ is used in the verification, we can compute this value directly in $Z/(M-1)Z$ by replacing from the beginning B_h and P_h by their projections, respectively denoted b and p, into $Z/(M-1)Z$, which are single-word integers. Fig. 4.3.2 shows the final binary multiplication algorithm with verification of the result using the proof by $2^m - 1$, in which we have yet more simplified the notations by replacing P_l by P and B_l by B.

```
1. mulmod( (A,a), (B,b), (N,n) )                    1.
1. {                                                2.
1.   p ← 0; P ← 0;                                  3. // Initialize
1.   a' ← 0 ; a'' ← 1 ;                             4. // Init eval of P(A)
1.   for( i ← 0 ; i < |A|_2 ; i ← i+1 ) {           5. // Loop on bits of
                                                       A
1.     if( A^[i(2)] ≠ 0 ) {                         6. // Addition
                                                       necessary?
1.       p ← p+b; P ← P+B;                          7. // Add B to P ,
2.                                                     then
if( P ≥ N ){ p ← p+1; P ← P−N ;}                    8. // correct modulo
                                                       N ,
3.       if( p ≥ M−1) p ← p−(M−1);                  9. // then modulo
                                                       M−1.
1.       a' ← a'+a'' ;                              10.// Eval P(A), then
2.       if( a' ≥ M−1) a' ← a'−(M−1) ;              11.// correct modulo
                                                       M−1
1.     }                                            12.
1.     a'' ← a''+a'' ;                              13.// Double a'', then
2.   if (          a'' ≥ M−1          )              correct
a'' ← a''−(M−1) ;                                   14.// modulo M−1
1.     b ← b+b; B ← B+B;                            15.// Double B , then
2.     if(          B ≥ N          )                correct
{ b ← b+1; B ← B−N ; }                              16.// modulo N , then
3.     if( b ≥ M−1) b ← b−(M−1);                      modulo
                                                    17.// M−1
1.   }                                              18.
1.   Check that a ≡ a' ;                            19.// Final check on
                                                       tests
1.   Check that a ⊛ b ≡ p ⊛ n ⊕ P ;                 20.// Final check
1.   return (P_I, P(P_I)) ;                         21.// Return both real
                                                       and
2. }                                                22.// reduced results.
```

Fig. 4.3.2 : binary modular multiplication with verification.

4.4 Computational overhead

The computational overhead consists in:

1. the precomputation of the projections of the operands and of the modulus, the result and each of the parameters consist of two fields: the number itself and its projection in $Z/(M-1)Z$;
2. one single-precision addition is added to each multiple-precision operation, with possibly a reduction modulo $M-1$;
3. the final checking test implies one multiplication in $Z/(M-1)Z$.

Out of these sources of overhead, a single one is significant: the added single-precision operands. If the operands are 1024-bit wide and the word size is 32 bits, each *big integer* consists of 32 words. Thus the multiple-precision addition consists of 32 additions and, consequently, as the overhead consists of a single operation, its relative value is $1/32$.

4.5 Evaluation of security

We already know that a random error is detected with probability $1-\frac{1}{M-1}$. However can an induced fault generate a non-random error, i.e. an error which would preserve the result of the final test?

Obviously no arithmetic operation operates at the same time on the real values and on the reduces ones. Only the tests will have an effect on both.
A single error, which changes a value x into x' such that $x' \equiv x \bmod (M-1)$ will not be detected. However the only pair of numbers in $[0:M-1]$ which are congruent modulo $(M-1)$ are 0 and $(M-1)$. A single error on a single bit will always be detected because 2^i cannot be a multiple of $(M-1)$ if $M=2^m$ for some m, which is always the case on a binary processor.

Artificially changing the result of the test on the bits of A will add (resp. subtract) $B \times 2^i$ for some i to (resp. from) the result. Thus, the error will be detected if $B \neq 0 \bmod (M-1)$, which happens with probability $1-\frac{1}{M-1}$ if the attacker cannot inject a specific B. Otherwise, a specific check must be done on this test and it is why we added the statements to eval $P(A)$ and check it against the value passed as a parameter together with A.

A perturbation of a reduction modulo N will change p the reduced value 1 and the real value by N in the reverse direction. As a real N will never be a multiple of M, the error will be detected.

A perturbation of a reduction modulo $(M-1)$ will also be detected for the same reason.

4.6 Modular exponentiation

This operation is basically a sequence of modular multiplications. If each of the multiplications is properly protected, there is a single waek point in the exponentiation: the test on the bits of the exponent. However, the reduced value of the exponent can again be directly computed from its value and computed from the actual results of the tests of these bits as the bits of A were protected in Fig. 4.3.2.

As the reduced modulo $(M-1)$ value of the result is returned at the same time as the actual result of the modular multiplication, the additional penalty for using the proof by 2^m-1 in a modular exponentiation is just the initial reduction of the number to be exponentiated. In the case of the Diffie-Hellmann protocol\cite{DH76}, even this number is constant.

5. CONCLUSION

From a simple arithmetic trick, the *proof by 9*, and a property of rings with a unit, we have constructed coherent schemes, based on the *proof by* 2^m-1, to protect the binary implementation of the modular multiplication and the modular exponentiation from fault injection.

Even if the math behind them is relatively simple, these schemes will resist any single-bit fault and a random fault will have an exponentially low probability of not being detected.

Thus the *proof by* 2^m-1 is thus a cheap way to render the attack by fault injection very chancy if the builtin arithmetic is 8-bit or 16-bit and impractical if the builtin arithmetic is 32-bit or 64-bit.

Besides its cost is negligible in front of the cost of multiprecision computations in case of a software implementation and, in case of a hardware implementation, it requires very little additional hardware.

Further work, to be published soon, will extend this protection to the multiplication of Montgomery.

REFERENCES

[Boneh at *al.*, 1977] Boneh, Dan, DeMillo, Richar A. and Lipton, Richar J. (1977) On the importance of checking cryptographic protocols for faults. *LNCS*, 1233:37-51.

[Diffie and Hellmann, 1976] Diffie W. and Hellman M.E. (1976) New directions in cryptography. In *IEEE Transations on Information Theory*, volume 22, pages 644-654.

[Kocher et *al.*, 1999] Kocher, Paul C., Jaffe, Joshua and Jun, Benjamin (1999) Differential Power Analysis. *LNCS*, 1666:388-397.

[Kocher et *al.*, 1996] Kocher, Paul C., Jaffe, Joshua and Jun, Benjamin (1996) Tioming Attacks on Implementations of Diffie- Hellman, RSA, DSS and other systems. *LNCS*, 1109:104-113.

[Noufal and Nicolaidis, 1999] Noufal, I. Alzaher and Nicolaidis M. (1999). A CAD Framework for Generating Self-checking Multipliers Based on Residue Codes. In *Date'99*, pages R122-129.

[Quisquater and Samyde, 2001] Quisquater J.J. and Samyde D. (2001) Electromagnetic Analysis (EMA) measures and counter- measures for Smart Cards. *E-smartcard Programming and Security, I. Attali and T. Jensen, editors*, 2140:200-210.

[Rivest, Shamir and Adleman, 1978] R.L.Rivest, A.Shamir and L. Adleman (1978) A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120-126.

[Siefert, 2002] C. Siefert, C. Aumüller, P. Bier, W. Fischer, P. Hofreiter and J.P. (2002) Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures. In *LNCS-CHES 2002*, vol. 2523, pp 260-275.