Data Integrity in Hardware for Modular Arithmetic

Colin D. Walter

Computation Department, UMIST
PO Box 88, Sackville Street, Manchester M60 1QD, UK
www.co.umist.ac.uk

Abstract. An increasing mass market for cryptographic products leads to greater pressure on companies to fabricate chips which will recover from, and correct, sporadic errors resulting from design and fabrication faults, inadequate testing, smaller technology, ionising radiation, random noise, and so on. Where encryption is subject to such errors, large quantities of data can become totally corrupted or inaccessible unless fault detection is an integral part of the hardware arithmetic. Here realistically cheap methods are examined for checking the correctness of the arithmetic computations which are the basis of the RSA cryptosystem and Diffie-Hellman key exchange. As in ordinary integer multiplication, a modular residue checker function is used to detect errors and trigger re-computation when necessary. The mechanism will also detect most permanent faults. Some suggestions are made on how to correct infrequent errors without using additional hardware.

Key Words: *Computer arithmetic, cryptography, RSA, modular multiplication, modular exponentiation, soft errors, error correction, fault tolerance, checker circuit, testing, correctness, data integrity, Montgomery multiplication.*

1 Introduction

Mass production of embedded cryptographic systems is fast approaching for applications ranging from electronic purses and e-commerce authentication to secure mobile video telephony. Chip technology for these has advanced to the point where random effects, such as noise and ionising radiation, are already causing so many errors that the aerospace industry regularly performs computations three times and takes a majority decision [1]. Indeed, some attacks on cryptosystems involve the introduction of such transient hardware errors to perform differential fault analysis [3]. But faults can occur at any point in the process from design to fabrication as well as during operation. Consequently, as with other products, incorporation of fault tolerance methods should mean increased yield from chip fabrication, less expensive testing and higher customer satisfaction during operation. The disaster with the Pentium division algorithm [2] illustrates the company critical issues of releasing faulty products even when errors are extremely rare. So, in the light of such experience, it has been suggested that checking should become an integral part of all arithmetic operations beyond those with the simplest implementations [2].

Standard error correction coding techniques are not generally applicable to arithmetic operations. So incorrect functioning of the ALU cannot usually be detected this way. Moreover, whilst all 32-bit operations might be fully tested before each unit is shipped, this is not realistic for the larger co-processors which might soon be employed in a typical RSA implementation. Nevertheless, excellent test suites can still be built for RSA hardware [10]. Duplication and triplication of hardware for non-safety critical fault recognition is too expensive, and in any case does not solve design faults.

Whilst any error will almost certainly generate random junk which is immediately detected on decryption, it is not always easy to signal this and request the recomputation, especially when this then invokes two way communication between the parties involved. Indeed, storing incorrectly encrypted data or session keys on disk or smartcard memory may not be detected for some time. In the case of message signing, the inverse process of signature verification is often a relatively cheap way of checking the computation [8], §3. However, with RSA encryption [9], checking by decrypting (a large exponent) requires knowledge of a secret key, which may not be available, and is also much more expensive than the encryption (a small exponent). So this form of verification is generally impossible or uneconomic. Furthermore, it is well understood that the consequent re-encryption of the same data after a glitch can leak secret data from an embedded system [3]. Thus, correctness should be verified before any output is released and an identical recomputation avoided in making any correction.

The aim of this paper is to consider much more cost effective alternatives than decrypting everything or duplicating hardware. We first show how to apply a cheap residue check which, with high probability, will find any intermittent or random arithmetic fault. We will argue that it will also detect other errors caused by permanent physical and logical flaws which have passed unnoticed during design, production and testing or which develop during use. We then describe how to correct such errors by modifying arguments in such a way as to avoid performing the same flawed calculation again. The efficacy of the check is discussed as well as the checking frequency. We conclude with an assessment of the time and area costs of the method.

2 Notation

The RSA algorithm [9] uses a public modulus M which is the product of two large primes, typically of around 2^9 bits each. For keys d and e, encryption of plain text T in the range $[0, M-1]$ and decryption of cipher text C are defined by $C = T^e \bmod M$ and $T = C^d \bmod M$ respectively. One of the keys d, e is kept secret, and the two satisfy the property $de \equiv 1 \bmod \phi(M)$ where ϕ is Euler's totient function. The strength of the system depends on the difficulty of factorising M, which is required in order to deduce one key from the other.

Hardware implementations of the cryptosystem often use a high radix or base for representing numbers. Typically this is a power of 2 such as 2^{16} or 2^{32} corresponding to the size of multiplier available. Let r denote this radix and n the number of base r digits in the modulus M. We will not encounter numbers larger than rM, so that a number A always has a representation

$$A = \sum_{i=0}^{n} a_i r^i$$

(The extra top digit may be required because occasionally numbers greater than M are encountered, in particular, just prior to modular reductions.) Exponentiation is performed by repeated modular multiplication, which in turn is performed by repeated modular addition. Thus the key operation is calculating products $P = (A\times B) \bmod M$ using a close relative of the following standard algorithm:

Classical Modular Multiplication Algorithm:

```
P <- 0 ;
For i <- n downto 0 do
Begin
   P  <- rP + a_i B ;
   q_i <-  P div M ;
   P  <-  P - q_i M ;
End
{ Post-condition: P = (A×B) mod M }
```

The initially generated sequence of digits q_j ($j = n$, $n-1$, ..., i) can be formed into an integer $Q_i = \sum_{j=i}^{n} q_j r^{j-i}$ and the initially consumed digits of A form a similarly defined integer A_i. Then it is easy to verify by induction that $P = A_i \times B - Q_i \times M$ and $0 \le P < M$ are invariants which hold at the end of each iteration of the loop. Hence the given post-condition holds when the loop terminates and, for $Q = Q_0$,

$$P = A\times B - Q\times M \tag{1}$$

Some dedicated hardware implementations of RSA with small radix r (typically $r = 2$ or 4) provide combinational logic circuitry for the equivalent of a complete modular addition cycle

$$P \quad \leftarrow \quad rP + a_i B - q_i M \tag{2}$$

Then, for speed, only an approximate value for q_i is used and this is calculated in advance from P. It is sufficiently accurate to keep P less than a small multiple of M, often $2M$ or rM. So a small, final modular subtraction may be necessary to obtain a result P in the range $[0, M-1]$. If we assume this extra modular correction is incorporated into P and Q then their final values still satisfy (1) and Q is again the integer quotient $(A\times B)$ div M.

There is a widely used alternative algorithm due to P. Montgomery [7] which processes the bits of A in the opposite order with a shift of P downwards instead of upwards. The advantage of this is primarily in hardware implementations rather than in software: successive modular reductions can commence without waiting for carries to propagate over the full length of the adder. The algorithm is the following, but it computes a shifted modular product instead, namely $(A\times B\times R^{-1}) \bmod M$ where $R = r^{n+1}$.

Montgomery's Modular Multiplication Algorithm:

```
P <- 0 ;
For i <- 0 to n do
Begin
    q_i <- (P + a_i B)(-M^-1) mod r ;
    P   <- (P + a_i B + q_i M) div r ;
End
{ Post-condition: P ≡ (A×B×R^-1) mod M for R = r^(n+1) }
```

For $M^{-1} \bmod r$ to be defined properly, we require M to be prime to r. Invariably, r is a power of 2 and M is odd, so this is not a significant restriction. Observe that the definition of q_i means that the division by r is exact. Hence $A\times B$ is computed, reduced by a multiple of M, and shifted by R. It is easy to obtain a bound on the size of the output P, e.g. $P < B+M$, which shows that it is the least non-negative residue $(A\times B\times R^{-1}) \bmod M$ to within a known, very small multiple of M [11]. The $\bmod r$ operation is fast because it only depends on the lowest digits of M, B and P, and the $\operatorname{div} r$ operation is fast because it only involves a hardware shift.

As with the classical algorithm above, the initially generated sequence of digits q_j ($j = 0, 1, ..., i$) can be formed into an integer $Q'_i = \sum_{j=0}^{i} q_j r^j$ and the initially consumed digits a_j ($j = 0, 1, ..., i$) of A form a similarly defined integer A'_i. Then it is easy to verify by induction that

$$P = (A'_i\times B + Q'_i\times M)/r^{-i-1} \tag{3}$$

is an invariant which holds at the end of each iteration of the loop. Taking $Q = Q'_n$, when the loop terminates,

$$P\times R = A\times B + Q\times M \tag{4}$$

So the post-condition holds. (The analogy with (1) is that Q is an r-adic approximation to the quotient $(-A\times B)/M$.) A small, final modular subtraction may be necessary to obtain a result P in the range $[0, M-1]$. If we assume this extra modular correction is reflected in a corresponding update to Q, then the final values of P and Q still satisfy (4).

3 A Simple Check for Soft Errors

A standard choice, [5] §7, for a checker function f in integer arithmetic is

$$f(A) = A \bmod D \tag{5}$$

where $D > 1$ is a suitable small number prime to at least $2r$, such as 15. This function is easily computed but fails to commute with the arithmetic operations of modular arithmetic. Ideally, for the arithmetic operation $\otimes$ which we wish to check, what is needed is a function f from integers mod M to integers mod D with the property

$$f(A\otimes B) = f(A) \otimes f(B)$$

for residues A and B in the ring of integers mod M. However (5) fails to have this property unless D divides M. The solution is to go back to the non-modular integers that the machine uses for its representation and take into account the modular subtractions made by the system. So, if P is the integer representing the result of the calculation of $A\otimes B$ during which Q subtractions of M are made, then

$$P = A\otimes B - Q\times M \tag{6}$$

The function f of (5) can be applied to this integer relation to obtain that

$$f(P) = f(A)\otimes f(B) - f(Q)\times f(M) \tag{7}$$

holds mod D if all the calculations involved have been performed correctly.

This applies to any modular arithmetic operation $\otimes$ from addition to exponentiation and, in particular, to modular multiplication. From here on we will interpret $\otimes$ as the particular modular multiplication operation of interest to us. [1] So (6) translates into (1) or (4). These equations re-phrase the output of the multiplication process entirely in terms of non-modular arithmetic operations and, as stated, enable the checker function f to be applied. Then the main property (7) to check becomes, respectively,

$$f(P) \equiv f(A)\times f(B) - f(Q)\times f(M) \bmod D \tag{8}$$

or

$$f(P)\times f(R) \equiv f(A)\times f(B) + f(Q)\times f(M) \bmod D \tag{9}$$

A difference between the left and right sides guarantees an error somewhere (although perhaps in computing f rather than $\otimes$) and, conversely, we will see that agreement is rare when the computation of $A\otimes B$ does contain an error.

[1] In *"Method and apparatus for protecting public key schemes from timing and fault attacks"* (US patent 5,991,415, Nov 23, 1999), Adi Shamir recommends obtaining and checking $A^e \bmod M$ by computing $A^e \bmod MD$ first, reducing this mod M for the result, and reducing it mod D to check against $(A \bmod D)^e$. This avoids computing $Q \bmod D$. Similarly, any operation might be performed mod MD and then reduced mod M for the result and mod D for the check.

4 The Choice of Modulus D

What is the best choice for D? The smaller D is, the cheaper and easier it is to compute the function $f = f_D$. However, we need to analyse the different possible faults to see how large D has to be to give the required degree of confidence in the correctness of the calculations. It turns out that almost all the hardware can be protected against a single fault with a very reasonable value for D.

To deal with register stuck-at faults, D should divide by a prime which does not divide $2r$. For most number representations likely to be used, any single bit error in the input to f changes that input by a number of the form $2^i r^j$. So, by the divisibility condition, this will be reflected in a different output value for f. Suppose the stuck-at fault is in register P and that register is written to, but not read from, during the multiplication. Then the left side of (7) will be incorrect when the faulty bit is stuck at the wrong value. So it will differ from the correct value computed for the right side. Hence this fault will be caught whenever it occurs, which will be in 50% of all cases on average. Of course, once an error is read from P, errors will start propagating further.

A similar argument applies to register M when it has a stuck-at fault. However, in this case all multiplications in an exponentiation are done correctly if the bit is stuck at the value which M should have, or they are all incorrect if the bit is stuck at the wrong value. Since $f(Q)$ will be 0 in $1/D$ of all cases, the equation (7) will not detect an error every time one occurs. However, over a single exponentiation which involves at least several multiplications, $f(Q)$ is unlikely always to be 0. So, if every multiplication is checked, the error should eventually be detected during the exponentiation, providing the calculation of $f(M)$ is based on the value of M kept in memory rather than the value in the faulty register used by M during the modular multiplication. Indeed, the possibility of errors in copying from and writing to memory illustrates the benefit of storing the checker function value with the number itself, in the same way as a parity bit.

Most registers used by a modular multiplication, apart from that holding M, will be both written to and updated a number of times, resulting in a propagation of errors. One might reasonably assume that this leads to the values on the left and right sides of (7) being essentially independent, so that $1-D^{-1}$ of all errors in multiplications are detected. (The undetected cases arise from the value in error being multiplied by 0.) Consequently, virtually all incorrect exponentiations will be detected, especially if each multiplication is checked, and permanent faults will be detected with greater probability than transient faults because more checks may contain the error.

A similar argument applies for faults in the combinational logic of a digit slice of an adder used to perform (2) or the equivalent step in Montgomery's method. The adder has three inputs, of which B and M are scaled by a digit and B may have a redundant form. At the level of the jth digit slice, the equation for the classical algorithm is

$$p_j + r \times c_{out} \quad \leftarrow \quad p_{j-1} + a_i \times b_j - q_i \times m_j + c_{in} \tag{10}$$

where c_{in} and c_{out} are carries from/to neighbouring slices and b_j and q_j may have redundant forms. Hardware for computing this may be repeated for every digit, or instead there will be a digit multiplier and adder which is reused for each digit position. For convenience, let us ignore the negative sign and assume all quantities are positive. (In practice there is a borrow to achieve this.) Typically the non-redundant digits might be bounded above by $r-1$, the redundant digits by $2r-1$ and the carry by $4r-2$. Then the expression on the right is bounded above by $4r^2-r-1$, which splits into a non-redundant digit of P and a carry still bounded by $4r-2$. Thus each line into, or out of, the combinational logic of the jth digit slice typically represents a value dr^j where d is a small power of 2 equal to, or less than, $2r^2$. Then summing the output values for all lines will give a total bounded above by $4r^2-1$. In this case, any error within the slice will make an absolute difference to the output also of the form dr^j where now $d < 4r^2$. Our desire is that any such difference should make a non-zero change to $f(P)$, i.e. the change should not be divisible by D. Thus any D larger than and prime to $4r^2$ is acceptable as it will detect all such single errors. In general, whatever the circuitry and bounds on the digit values, any value larger than the sum of all digit slice output lines would do for D. If some output values d cannot arise without multiple errors, a smaller choice for D might well be possible. All such possible values of d can easily be determined from the circuit design before fabrication, and the tendency will be for d to be a multiple of 2 times a small odd number.

The digit slice error may propagate in two ways, depending on whether it is transient or not. With a permanent fault, a substantial proportion of the addition cycles are likely to be affected in the same way. As RSA multiplications contain many addition cycles, $f(P)$ is most likely to change in a way which makes the differences between correct and incorrect values uniformly distributed $\bmod D$, even although they may all be multiples of the above d. Then the checker function will detect all but $1/D$ of the errors which occur. However, with a transient fault, the difference between the correct and computed values of P is shifted up or down by a power of r on each iteration. So its initial primeness to D is preserved. Eventually the error may affect the value of Q, but there will be a compensating deduction of a multiple of M from P which will not obscure the difference between the values of the left and right sides of (7). So such errors should always be spotted.

The rest of the combinational logic includes counters, clocks, control circuitry, etc. These subcircuits take less area than the multiplier or digit slices and could mostly be checked by duplication. However, errors there will tend to have a random effect on the outputs, yielding approximately a $1/D$ probability of the residue check falsely approving an incorrect calculation. Hence employing a large D could be an alternative to duplicating such hardware. The main exception is the exponentiation circuitry. Although this controls the sequence of multiplications and so cannot affect the truth of (7), it is usually implemented in software. So this remains unchecked because f only checks hardware arithmetic operations.

Finally, there may be specialised hardware for computing digits of Q. In most cases an error in a digit of Q will either lead to overflow/underflow because after several more iterations during which P is shifted, P will grow too large or become negative. Alternatively, the self-correcting nature of the choice of q_i will successfully compensate for the error. Either way, the possibility of over- or under- flow must be monitored because the equation (7) will not detect such errors: the compensating multiple of M will re-adjust the equation so that it still holds. So a final range check on P might not come amiss.

In summary, most of the hardware is protected against transient and permanent faults by the checker function. When typical redundant representations are used, errors are detected except in at most $1/D$ of cases if we are allowed to choose $D > 4r^2$ and prime to $2r$. For compatibility with the hardware multiplier, it is clearly advantageous to keep $D < r$, which is the built-in size of all non-redundant digits. The arguments above suggest that taking a large $D < r$ with some large prime factors would achieve most or even all of our requirements, its only disadvantage being to limit the probability of detecting some errors. This is efficiently held as a single digit, so we will assume such a choice is made for D. Other alternatives might be to pick a large two-digit D, i.e. one which is less than r^2, or even to use two co-prime values of D, each just less than r. This might be preferred for very small r (such as 2) to retain good detection rates.

5 Time and Area Costs for Checking

The choice of D has implications for the cost of computing f. However, since the processor cycle time is probably determined by the multiplier, it is likely that digit sums and digit products are computed in essentially the same time. We will assume r is a power of 2 and look at two possibilities.

First, suppose D is a divisor of $2^s \pm 1$ for some s. This is the standard situation analogous to the case of checking divisibility of a decimal number by 3, 9 or 11. Suppose A has a standard, non-redundant, binary representation. Then computing $f(A)$ simply requires computing the (possibly alternating) sum of s-bit digits of A (and perhaps repeating this on the result) and then reducing the result $\bmod D$. An obvious choice here is $D = r-1$, for which the digits of A are summed. The result for typical RSA implementations will be a two digit number whose digits are then themselves summed. If the result overflows one digit, D is subtracted by adding 1 to the lower digit to yield a single digit for $f(A)$ after $n+2$ additions overall. Without adding extra, dedicated hardware, taking $D = r-1$ is arguably the most economic solution. If A has a redundant representation, the extra bits must be added into the calculation in the same way, and this may double the number of additions required to obtain $f(A)$.

In general, computing $f(A)$ for some argument A can be performed iteratively from the most significant end using

$$f(A_i) = (f(A_{i+1}) \times f(r) + a_i) \bmod D \tag{11}$$

and $f(A_{n+1}) = f(0) = 0$, or computed similarly from the least significant end using $f(r^{-1})$. For the above choice of $D = r{-}1$, we have $f(r) = \pm 1$ so that the multiplication is avoided.

Alternatively to this choice, suppose a large $D{<}r$ is chosen for which $f(r)$ or $f(r^{-1})$, as appropriate, is small. Prime choices for D might be $r{-}16{\pm}1$ for $r = 2^{16}$ or $r{-}5$ for $r = 2^{32}$. Assume, in fact, that $f(r)^2(f(r){+}1) < r$. By leaving most of the reduction mod D in (11) to the end, we can obtain $f(A_i) < (f(r){+}2)r$ for each i by expressing $f(A_i) = m_i r{+}l_i$ as a two digit number, where $m_i \le f(r){+}1$ and computing $f(A_i) = m_{i+1}{\times}f(r)^2 + l_{i+1}{\times}f(r) + a_i$ instead. This converges and yields $f(A) \equiv m_0{\times}f(r) + l_0 < 2D$ if D is large enough, so that one more subtraction of D gives $f(A)$. The cost of computing f therefore amounts to $2n{+}2$ digit multiply-accumulate operations in this case.

In the context of RSA, $f(M)$ need only be calculated once for a given modulus. Besides this and the exponent, the only other input to an exponentiation is the initial text T for which $f(T)$ must be calculated. Thereafter, for each multiplication, only the check values for the outputs need to be calculated, namely $f(P)$ and $f(Q)$. For the more expensive of above choices, this adds $4n + 4$ digit multiply-accumulate operations to the $2n^2$ required for a full length modular multiplication using (10). A further 3, resp. 4, such operations are required to check (7) via equations (8) and (9) respectively. So adding the checker function should be equivalent to adding at most 1 to the number of digits in M. By including another multiplier in an array of multipliers [11], [6], or extra cycles when there is a single multiplier, the cost can normally be spread over time and area so that both the time and area formulae reflect the increase in n by at most 1.

Finally, for very small values of r, such as $r = 2$ or 4, RSA hardware implementing (10) consists of a full length adder and no multiplier. Then a D of the order $4r^2$ is more appropriate than $D = r{-}1$. Computing $f(A)$ is straightforward using only additions so that the clock speed is maintained, but more digit additions are required. So, the work resulting from including the checker function corresponds to adding several more digits to n. However, as n is also greater, the proportion of extra work is not increased. It is in fact dependent on the size of D and how well it matches the digit base r.

6 Recovering from Transient Errors

When an error is detected, it may be unwise to continue computations since an attack on the system may be in progress. The checker function can indeed be used to defeat some attacks which operate by inducing transient errors. However, we will assume the system wishes recomputation to be performed. If errors are rare enough it is reasonable to cancel the exponentiation and just start again. This requires a single extra buffer for storing the original text T until the encryption/decryption has been approved. If the checking needs to be done on every multiplication, then, for most exponentiation schemes, it is the output of the previous multiplication which forms the only new argument to the multiplication which is in error. Thus, again, a single input needs to be buffered until the check is complete.

Since $f(Q)$ can be computed digit serially as its digits are generated, any error can be detected immediately $f(P)$ becomes available. With such large numbers, $f(P)$ would normally also be computed digit serially and is therefore not available until some time after P, unless P is also generated digit serially.

Suppose the modular multiplier uses redundancy to allow parallel digit operations on an array of multipliers and one modular multiplication starts immediately upon termination of the previous one. This is the classical model described by E. Brickell [4]. Now $f(P)$ can be computed using (11) and the check (7) just completed in the time to set up and perform the next modular multiplication. When an error is discovered, two modular multiplications must be discarded, namely the current one and the previous one for which the test has just detected an error. So, by buffering the new input of the current and previous modular multiplications so that such steps can be repeated when necessary, the exponentiation can proceed and be checked with a time penalty equivalent to $2k{+}1$ extra modular multiplications where k is the number of multiplications containing detected errors. On average, one would expect k to be very close to 0.

More recently, systolic and linear arrays have been combined with Montgomery's algorithm to provide modular multipliers [11], [6]. These avoid some of the drawbacks of the standard design, such as redundancy and digit broadcasting which have time and area penalties. So a slightly faster clock is possible. The arrays operate with digit serial I/O to the multiplier array and, by performing two streams of multiplications in parallel, can have the same throughput in terms of clock cycles despite the inherent problem of only being able to use cells on every alternate cycle. A single multiplication produces one digit only every other cycle, resulting in just over $4n$ time slots between the first digit input and last digit output. Now $f(P)$ can be computed as the digits of P are generated and the correctness check made within a single clock cycle of P being produced. In the case of the linear systolic array, it suffices to buffer the new inputs of the four multiplications currently in progress in the multiplier (one starting and one finishing in each of two interleaved streams) so that the ones just finishing can be recomputed if necessary. The buffers might even be shared between the two streams if the probability of a double error were sufficiently small.

Thus, in addition to the cost for detecting errors, the occurrence of random encryption/decryption errors can be corrected by recomputation with an area cost of only several full length buffers (the precise number being dependent on the implementation), and a time penalty of $2k{+}1$ extra modular multiplications where k is the number of detected errors. For the smallest radix, $r = 2$, the extra registers may easily double the total hardware area, but as r increases the proportion devoted to registers falls and the relative cost diminishes. However, this solution is still a much cheaper alternative than voting between three copies of the hardware, or using backup registers to enable re-computation when two copies of the hardware fail to agree.

7 Permanent Faults

We have now treated transient errors and seen how most of these can be successfully recognised and recovered from. Finally, permanent errors need consideration. Most design or fabrication faults should be caught during comprehensive production testing [10], but this is expensive and shortcuts are bound to lead to faulty products being delivered. Ideally, as a minimum, every combination of inputs should be tested (i) for every digit slice and (ii) for the computation of q_i. However, as the modulus M is not usually changed very frequently, some errors in the hardware logic may not surface at testing nor even occur during the chip's life.

Whilst the major test of correct encryption is that decryption does not yield rubbish, in RSA one key is always kept private so that such a test for a specific modulus may be denied. Thus, as encrypted text is indistinguishable from rubbish, some kind of on-board checking of output is desirable before destroying the plain text input.

The arguments already presented for detecting transient errors apply almost equally well to detecting permanent faults: the two are indistinguishable if the hardware at fault is only operated once. But, in general, we have already seen that repeated faults will cause (7) to detect all but at most $1/D$ of arithmetic and some other errors, but that logical errors in the computation of q_i are not discovered since both P and Q are affected equally. In particular, our experience of building previous chips suggests that the final adjustment to the last digit of Q which puts P into the interval $[0, M-1]$ is the most frequent cause of undetected errors, especially for P near a multiple of M. Such logical errors can be very infrequent. However, it is the correctness of the modular arithmetic which is the subject of this article. Such errors tend to keep recurring because the faulty hardware is either reused with the same values for every exponentiation or it is part of a digit operation which is executed a very large number of times with effectively random data. Hence they will almost certainly be detected.

Correction after transient errors is obtained simply by running the same hardware again with the identical inputs. Of course, this is useless for permanent errors. Instead, with the usual assumption that the errors are rare, rather than use alternative hardware which may contain the same design errors, the inputs can be modified in an attempt to avoid the errors.

An error with a particular digit slice might be avoided by a simple shift: $T^e \bmod M$ is computed via $T^e \bmod rM$, so that the combination of bits which cause the error might be avoided. This just requires a slight modification to the hardware or software which makes the final modular correction to bring the output of the modular multiplier into the correct range $[0, M-1]$. A bigger shift might avoid the use of the faulty digit slice entirely. Of course, this form of adjustment is not an option when Montgomery's method is used since the new modulus must stay prime to r, nor will it work if the same hardware is used for every digit position. Then, or if the problem is with the most or least significant digits of M, a similar solution of computing $T^e \bmod dM$ for a digit d prime to r may succeed. Other inputs than M to the digit operations all vary so much and so frequently that the digit combination expressing the error will arise very frequently whatever input modifications are made. This is enough to make disposal and replacement of the chip the best solution.

8 Summary and Conclusion

The detection and correction of transient errors in a hardware implementation of the RSA cryptosystem is straightforward to implement and can be used to defeat certain types of active attack on embedded systems such as in smartcards. It can be done efficiently and reliably with acceptable time and area costs equivalent to an increase in the size of the modulus by one digit or less plus some extra buffering. Successful correction must usually assume the correctness of the hardware. However, the checker function and other outlined methods will also detect most logic errors and fabrication faults as well as transient ones. With minor extra work which could be supplied by software, these too might be corrected if they are sufficiently infrequent.

Incorporating a checker function such as (5) and keeping an eye out for overflows are increasingly essential with shrinking technology and may prevent the loss of considerable data when an error inevitably strikes.

References

1. J. M. Benedetto, "Economy-class Ion-defying ICs in Orbit", *IEEE Spectrum*, vol. 35, no. 3, March 1998, pp 36-41
2. M. Blum and H. Wasserman, "Reflections on the Pentium Bug", *IEEE Trans. Comp.*, vol. 45, no. 4, April 1996, pp 385-393
3. D. Boneh, R. DeMillo and R. Lipton, "On the importance of checking cryptographic protocols for faults", *Eurocrypt '97*, Lecture Notes in Computer Science, vol. 1233, Springer-Verlag, 1997, pp 37-51
4. E. F. Brickell, "A Fast Modular Multiplication Algorithm with Application to Two-Key Cryptography", *Advances in Cryptology - CRYPTO '82*, Chaum et al., Eds., New York, Plenum, 1983, pp 51-60
5. G. Gerwig and M. Kroener, "Floating Point Unit in standard cell design with 116 bit wide dataflow", *Proc 14th IEEE Symposium on Computer Arithmetic*, Adelaide, 14-16 April 1999, IEEE Press, 1999, pp 266-273
6. P. Kornerup, "A Systolic, Linear-Array Multiplier for a Class of Right-Shift Algorithms", *IEEE Trans. Comp.*, vol. 43, no. 8, April 1994, pp 892-898
7. P. L. Montgomery, "Modular Multiplication without Trial Division", *Math. Computation*, vol. 44, 1985, pp 519-521
8. J.-J. Quisquater and M. De Soete, "Speeding up smart card RSA computations with insecure coprocessors", *Proc. Smart Card 2000*, D. Chaum editor, Elsevier Science, 1991, pp 191-197
9. R. L. Rivest, A. Shamir and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", *Comm. ACM*, vol. 21, 1978, pp 120-126
10. C. D. Walter, "Moduli for Testing Implementations of the RSA Cryptosystem", *Proc 14th IEEE Symposium on Computer Arithmetic*, Adelaide, 14-16 April 1999, IEEE Press, 1999, pp 78-85
11. C. D. Walter, "Systolic Modular Multiplication", *IEEE Trans. Comp.*, vol. 42, no. 3, March 1993, pp 376-378