Optimised Squaring of Long Integers Using Precomputed Partial Products

Braden Phillips
Electronic Engineering Division, Cardiff University
b.phillips@computer.org

Abstract

This paper considers the combination of two familiar, but hitherto incompatible, arithmetic techniques: optimised squaring and precomputing partial products.

Optimised squaring reduces the total accumulation effort required for squaring when compared with multiplication, by removing repeated digit products from the accumulation tree.

Iterative implementations of integer multiplication, in which each partial product is evaluated and accumulated in turn, can often be accelerated by precomputing the set of partial products and accumulating these as required. Iterative implementations of optimised squaring cannot benefit from the same straightforward technique.

In this paper a new algorithm for optimised squaring is developed which reconciles the these two techniques and which is an improvement over squaring by multiplication for some platforms. The result is of significance for the implementation of public key cryptography on smart cards or other small footprint devices.

1. Introduction

The implementation of most public key cryptosystems involves the evaluation of modular powers such as $A^B \bmod N$ where A, B and N are all long integers: often 1024-bits or more [1]. The exponentiation step is invariably implemented as a series of multiplications and squares and while techniques exist to reduce the number of multiplications, the number of squares is always approximately equal to the length of the exponent in bits [2]. It is not surprising, therefore, that squaring and modular squaring of integers has become a very important problem.

More and more frequently, the platforms required to implement these cryptographic functions are low power, small footprint devices such as smart-cards, mobile phones or PDAs. At such long wordlengths, and with limited hardware resources, a full array multiplier is not feasible. Instead, multiplication and squaring must be performed by multiprecision software or in an iterative fashion using a long hardware accumulator [3].

In this environment, *old* techniques such as multiplication using precomputed partial products and optimised squaring take on new relevance. These two techniques, described in more detail in the subsequent section, can each be used to accelerate long-integer squaring; however, they cannot be combined in any straightforward manner. In Section 2 a new means of combining these two techniques is developed. Section 3 reports on an implementation of this technique. Execution times are discussed for an assembly language implementation on an ARM6 processor without a hardware multiplier.

2. Problem definition

I consider squaring the binary integer:

$$A = \sum_{i=0}^{n-1} \alpha_i 2^i \text{ with } \alpha_i \in \{0, 1\} .$$

A sliding-window digit-set conversion is applied to A during the square such that A is converted to the representation:

$$A = \sum_{i=0}^{n-1} a_i 2^i .$$

The digits a_i belong to the redundant set of odd digits (and zero) $\{0, 1, 3, \ldots, 2^m - 1\}$ where m is the window size in bits. This representation is used instead of a non-redundant higher radix as the sliding window form yields fewer non-zero digits on average and hence reduces the number of non-zero partial products to accumulate (as applied to multiplication in [5]). The combination of optimised squaring with sliding windows is considered in [6]. The current paper takes this result a step further and incorporates precomputed partial products into the scheme.

The squaring algorithm developed here is intended for a platform capable of adds and shifts but without a hardware multiplier.

2.1. Multiplying with precomputed partial products

The multiplication $A \times B$ can be expressed as a sum of shifted partial products:

$$A \times B = \sum_{i=0}^{n-1} a_i B 2^i .$$

The partial products $a_i B$ can be computed as required, or precomputed once for all digit values and then accumulated. The latter scheme will be more efficient for long wordlengths and will represent a considerable saving if the generation of $a_i B$ is expensive (as it will be if there is no hardware multiplier).

For an add and shift based system, the partial products $\{B, 3B, \ldots, (2^m - 1)B\}$ can be precomputed and stored in $(2^{m-1} - 1)$ steps using the iteration $(i+2)B = iB + 2B$.

2.2. Sliding window conversion

Conversion to the redundant digit set $\{0, 1, 3, \ldots, 2^m - 1\}$ can be performed by scanning the bits of A starting with the least significant bit and progressing to the left. If the bit being scanned, α_i, is zero, then the converted digit a_i is also set to zero. If $\alpha_i \neq 0$ then a_i is set to the value of the m-bit string $[\alpha_{i+m-1} \ldots \alpha_{i+1} \alpha_i]$:

$$a_i = \sum_{j=0}^{m-1} \alpha_{i+j} 2^j$$

The subsequent digits a_{i+1} up to a_{i+m-1} are set to zero and scanning is resumed at α_{i+m}.

A non-redundant radix 2^m representation of A would have on average $n(2^m - 1)/(m \times 2^m)$ non-zero digits. The sliding window conversion results in $n/(m+1)$ non-zero digits on average.

Rather than store the converted representation of A it is more convenient and efficient to perform the conversion during the square: at iteration i the converted digit a_i is found and then the partial product $a_i A$ is accumulated.

2.3. Optimised squaring

Optimised squaring is based on the observation that when squaring using multiplication, most of the required digit products are repeated twice [4]. For example, in the evaluation of $A \times A$ the digit products $a_i a_j$ and $a_j a_i$ (where $i \neq j$) are both required but will be equal. Array implementations, or multiprecision software, can take advantage of this by accumulating the shifted digit product $2a_i a_j$. In this way the total number of digit products to evaluate and accumulate is approximately halved. In practice the speed-up obtained is somewhat less than two. In [7] the time for a square is 0.57 times that of a multiplication. In [8] the speed-up factor is between 0.7 and 0.85.

Figures 1, 2 and 3 show the development of an alternative that is more applicable to an architecture based on a long accumulator. The familiar *schoolbook* form of multiplication is shown in Figure 1. In Figure 2 the partial product terms in the tableau have been re-arranged so that repeated digit products are accumulated together. A further rearrangement of the tableau in Figure 3 demonstrates an optimised form involving the accumulation of long 'partial squares' $a_i A_{i+1}$ (defined below).

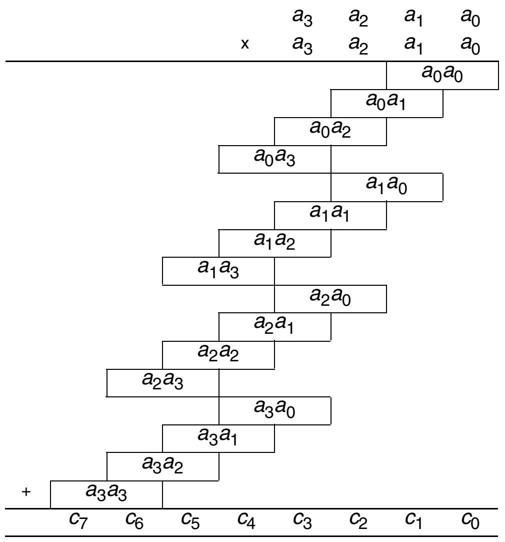


Figure 1. Operand scanning square $C = A^2$.

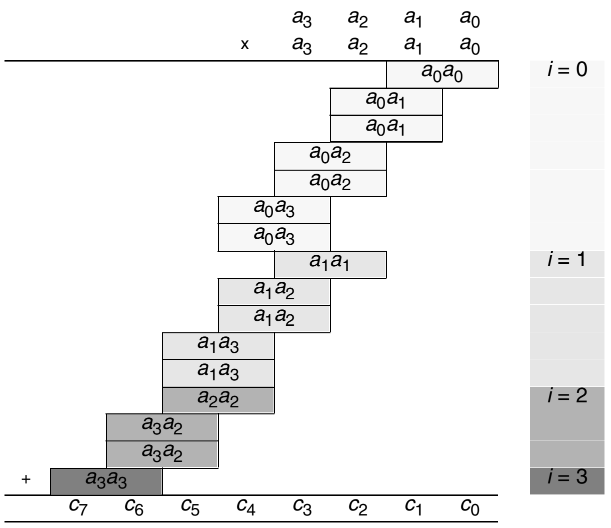


Figure 2. Re-arranged partial product tableau.

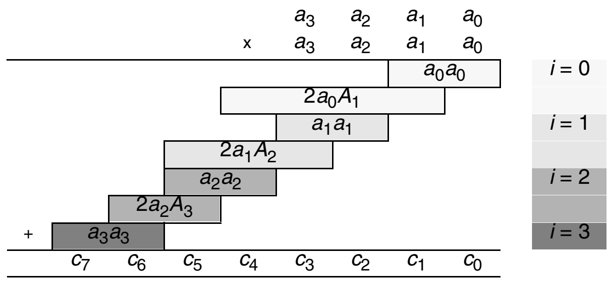


Figure 3. An optimised form of squaring.

The optimised square shown in Figure 3 can be described by the following iteration:

$$A^2 = \sum_{i=0}^{l-1} \left(a_i^2 r^{2i} + 2a_i A_{i+1} r^{2i+1} \right) \tag{1}$$

$$\text{where } A_i = \sum_{k=i}^{l-1} a_k r^{k-i} .$$

The term A_i is a truncated version of A obtained by shifting A by i digits to the right (and discarding any digits shifted to the right of the radix point). Equation 1 shows that at each iteration, A is truncated and a multiple of the truncated result, $a_i A_{i+1}$, is accumulated. Note that this form of optimised squaring, rather than reducing the number of partial products to accumulate, reduces the average length of the partial products; however, it is not obvious that the partial products $a_i A_{i+1}$ can be readily calculated from a precomputed, un-truncated value, $a_i A$.

Before proceeding it will be helpful to define some new terminology. In the subsequent development the terms $a_i A_{i+1}$ will be referred to as *partial squares*. The partial square jA_i will be given the symbol $A_{i,j}$. This is formed by taking the top $(n-i)$ digits of A and multiplying the result by j:

$$A_{i,j} = jA_i = j \times (A \gg i) .$$

Where multiples of A are precomputed, they will be called *precomputed multiples* and are given the symbol $A_{0,j}$:

$$A_{0,j} = j \times A \quad .$$

In the development that follows it is also important to retain the distinction between the original bits of a non-redundant binary representation of A and the redundant digits of A that result from the sliding window conversion. Recall that α_i is the ith bit of a non-redundant binary representation of A and a_i is the ith digit of A after digit set conversion.

2.4. Optimised squaring with precomputed partial products

The problem of combining optimised squaring, sliding window conversion and partial product precomputation can now be expressed for the new iteration (Equation 1) and the new notation. At iteration i of the square, the digit a_i is determined from the bits $\alpha_i \ldots \alpha_{i+m-1}$. A partial square is then formed from A shifted to the right and multiplied by a_i. The final step of the ith iteration is to accumulate the partial square into the partial result.

However, as discussed in the next section, there are two problems to be overcome with this naive formulation: the first is due to the changing representation of A during the square; and the second is in obtaining correct partial squares from the precomputed multiples.

3. Problems and solutions

3.1. Sliding windows and optimised squaring

At iteration i the sliding window conversion may change the value of the ith bit of A, α_i, to the digit a_i and correct for this change by adjusting the digits of A to the left. Thus the value of A_{i+1} also changes. In the new representation, the previous digits a_0 to a_{i-1} and the values A_0 to A_i remain unchanged; hence, the existing partial result from the previous iteration is valid for the new representation.

Figure 4 demonstrates the remaining problem: as the digit set conversion is applied to a_i, the value of A_{i+1} changes and if the partial square $a_i A_{i+1}$ was precomputed based on the original value of A_{i+1} then it will not be correct following the conversion.

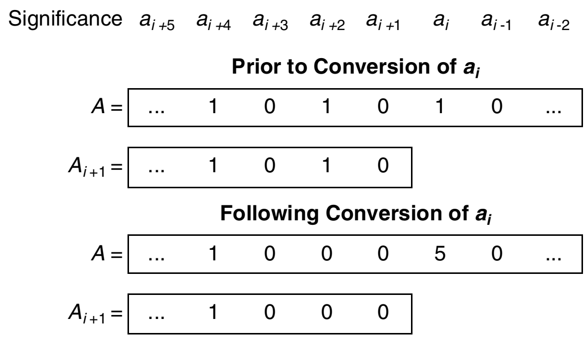

Significance a_{i+5} a_{i+4} a_{i+3} a_{i+2} a_{i+1} a_i a_{i-1} a_{i-2}
Prior to Conversion of a_i
$A =$ ... 1 0 1 0 1 0 ...
$A_{i+1} =$ ... 1 0 1 0
Following Conversion of a_i
$A =$ ... 1 0 0 0 5 0 ...
$A_{i+1} =$ ... 1 0 0 0

Figure 4. The value of A_{i+1} changes following conversion of a_i (using $m = 3$).

To overcome this problem note that when a non-zero digit a_i is selected, it is such that the subsequent digits $a_{i+1}, \ldots, a_{i+m-1}$ all become zero. None of the other digits in A_{i+1} are changed. Following the selection of a_i we have $A_{i+1} = 2^{m-1} \times A_{i+m}$ and A_{i+m} is unchanged by the conversion.

The iteration of Equation 1 can be modified so that iteration i accumulates the partial square $a_i A_{i+m}$. This has the advantage that A_{i+m} is unchanged since the beginning of the square. The modified iteration is:

$$A^2 = \sum_{i=0}^{l-1} \left(a_i^2 2^{2i} + A_{i+m,a_i} 2^{2i+m+1} \right). \qquad (2)$$

3.2. Obtaining partial squares from precomputed multiples.

Equation 2 shows that iteration i of the square accumulates the partial square A_{i+m,a_i} where this is formed from A shifted $(i+m)$ -bits to the right and then multiplied by a_i. However, as demonstrated in Figure 5, this is not the same as shifting the precomputed multiple A_{0,a_i} by $(i+m)$ digits to the right. If precomputation is to work then it must be possible to quickly determine the partial square A_{i+m,a_i} from the precomputed multiple A_{0,a_i}.

Figure 5 demonstrates the problem with combining precomputed partial products with optimised squaring: the value of the partial square $A_{7,5}$ is not a trivial truncation of the precomputed multiple $5A$.

The remaining question is: given a set of precomputed multiples $A_{0,j}$, is it possible to quickly determine the partial squares A_{i+m,a_i} ?

3.3. Solution

Consider the evaluation of the precomputed multiple A_{0,a_i} for an odd value of a_i from

$$A_{0,a_i} = a_i A_{0,1} = A_{0,1} + \left(\frac{a_i - 1}{2}\right) 2A_{0,1}$$

using the binary addition shown in Figure 6. In this diagram, the shaded cells show the addition required to determine A_{i+m,a_i}. Observe that:

$$A_{i+m,a_i} = \left(A_{0,a_i} \gg (i+m)\right) - \left(\frac{a_i - 1}{2}\right)\alpha_{i+m-1} - c_{i+m}$$
. (3)

In this equation the term $A_{0,a_i} \gg (i+m)$ is readily available from the table of precomputed multiples and given that we are using a sliding window conversion, the bit α_{i+m-1} can be determined from a_i. It remains to compute the unknown carry term c_{i+m}. Figure 7 shows the generation of this term. Examination of this figure gives the following equation for c_{i+m}:

$$= \frac{\left(a_i + (a_i - 1)\left(a_i \bmod 2^{m-1}\right) + \left(\frac{a_i - 1}{2}\right)\alpha_{i-1} + c_i\right)}{2^m}$$
(4)

In Equation 4, the term c_i is unknown and α_{i-1} is not readily available. Let us use $cmin_{i+m}$ to denote the minimum possible value for c_{i+m}. This occurs when $c_i = 0$ and $\alpha_{i-1} = 0$:

Significance	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
	Pre-computation													
$A = A_{0,1} =$	0	0	0	1	1	0	1	1	0	1	0	0	1	1
$5A = A_{0,5} =$	1	0	0	0	1	0	0	0	0	1	1	1	1	1
	Iteration *i* = 4 with *m* = 3													
$A =$	0	0	0	1	1	0	1	0	0	5	0	0	0	3
$A_{i+m} = A_7 =$	0	0	0	1	1	0	1							
$A_{i+m,a_i} = A_{7,5} = 5A_7 =$	1	0	0	0	0	0	1							
$A_{0,5} \gg 7 =$	1	0	0	0	1	0	0							

Note that these two shaded terms are not equal.

Figure 5. A problem: the required partial square is not a trivial truncation of a precomputed multiple.

$A_{0,1}$		...	α_{i+m+1}	α_{i+m}	α_{i+m-1}	...	α_i	...
$+ 2A_{0,1}$	...	α_{i+m+1}	α_{i+m}	α_{i+m-1}	α_{i+m-2}	...	α_{i-1}	...
...	...	...	...	...	...	...	...	...
$+ 2A_{0,1}$	...	α_{i+m+1}	α_{i+m}	α_{i+m-1}	α_{i+m-2}	...	α_{i-1}	...
(Carry Bits)	...	...	...	c_{i+m}	...	...	c_i	...
$= A_{0,a_i}$	...	d_{i+m+2}	d_{i+m+1}	d_{i+m}	d_{i+m-1}	...	d_i	d_{i-1}

Figure 6. A binary addition to find A_{0,a_i} from $A_{0,1}$ and the carry bits that are generated.

$$cmin_{i+m} = \left(a_i + (a_i - 1)\left(a_i \bmod 2^{m-1}\right)\right) \gg m \quad . \qquad (5)$$

Similarly, the maximum $cmax_i$ occurs when $\alpha_{i-1} = 1$, and $c_i = (a_i - 1)/2$ where the latter represents the maximum possible carry out from the addition of $(a_i + 1)/2$ binary terms. Thus:

$$cmax_{i+m} = \left(2a_i + (a_i - 1)\left(a_i \bmod 2^{m-1}\right) - 1\right) \gg m \quad . \qquad (6)$$

Consideration of Equation 5 and Equation 6 gives $cmax_{i+m} - cmin_{i+m} \leq 1$. Therefore, application of Equation 5 yields a value for c_{i+m} that is either correct or too small by 1.

Returning to Figure 7, observe that:

$$d_{i+m} = (\alpha_{i+m} + ((a_i - 1)/2)\,\alpha_{i+m-1} + c_{i+m}) \bmod 2$$

where d_{i+m} is the $(i+m)$ -th bit of A_{0,a_i}. Manipulation of this equation yields:

$$c_{i+m} \bmod 2 = d_{i+m} \oplus \alpha_{i+m} \oplus (\alpha_{i+1} \wedge \alpha_{i+m-1}) \quad . \qquad (7)$$

So Equation 5 gives a value for c_{i+m} that is too small by at most 1 and Equation 7 reveals if the correct value of c_{i+m} is odd or even. Combining these two determines c_{i+m} exactly.

Although the analysis has been quite involved, it leads to a useful conclusion: given the correction terms c_{i+m} and α_{i+m-1}, the partial square A_{i+m,a_i} can be quickly determined from the precomputed multiple A_{0,a_i}. The entire procedure for optimised squaring with unsigned sliding windows and precomputed partial squares is summarised below. Note that the two correction terms of Equation 3 have been combined into a single term t thus: $t = ((a_i - 1)/2)\,\alpha_{i+m-1} + c_{i+m}$.

1. Precompute the table of partial squares $A_{0,j}$ for $j = 1, 3, 5, \ldots, 2^m - 1$ where $A_{0,1} = A$ and $A_{0,j+2} = A_{0,j} + 2A_{0,1}$.
2. Precompute a correction table $t_{min}(j)$ for $j = 1, 3, 5, \ldots, 2^m - 1$ from:
$$t_{min}(j) = ((j-1)/2)\,(j \operatorname{div} 2^{m-1}) + (j + (j-1)\,(j \bmod 2^{m-1}))\operatorname{div} 2^m \qquad (8)$$
3. Apply the iteration from Equation 2. At iteration i, to find A_{i+m,a_i}, determine the correction term:
$t = t_{min}(a_i) + d_{i+m} \oplus \alpha_{i+m} \oplus (t_{min}(a_i) \bmod 2)$
where d_{i+m} is the $(i+m)$ -th bit of A_{0,a_i}. This gives the partial square:
$A_{i+m,a_i} = \left(A_{0,a_i} \gg (i+m)\right) - t$.

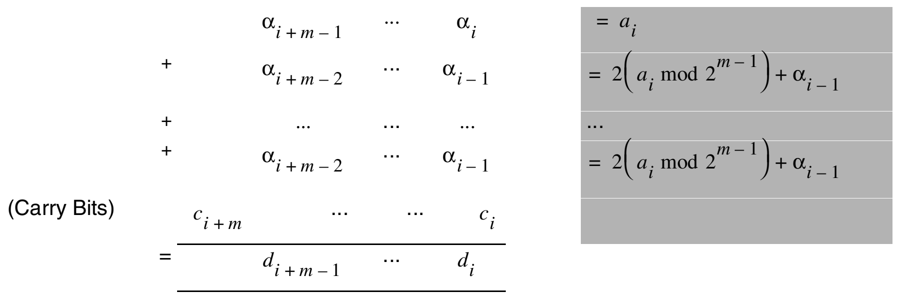
An example of an optimised square is provided in Figure 8.

4. Evaluation and conclusions

This optimised squaring algorithm was implemented in assembly language for an 25MHz ARM6 processor without a hardware multiplier. For 512-bit squares (as required for 1024-bit RSA using the Chinese Remainder Theorem), the optimal trade-off between precomputation and evaluation time occurred with $m = 6$. In this case optimised squaring took on average 796 µs and required 2304 32-bit words of memory storage for precomputed partial squares and temporary results. Further increases in window size increase the storage requirement and precomputation effort and result in an overall increase in average evaluation time.

Optimised squaring achieves a small improvement over multiplication for which the best average time was 863 µs for $m = 6$ and required 2176 words of data memory. In this case optimised squaring has improved the time over squaring using multiplication by a factor of 0.92.

Selection of smaller windows with $m = 5$ significantly reduces the storage requirement for a small loss of speed. In this case squaring takes 829 µs and requires 1216 words of data storage. Multiplication takes 895 µs and requires 1088 words of data storage. Here



α_{i+m-1} ... α_i
+ α_{i+m-2} ... α_{i-1}
+ ... ...
+ α_{i+m-2} ... α_{i-1}
(Carry Bits) c_{i+m} ... ... c_i
= d_{i+m-1} ... d_i
$= a_i$
$= 2\left(a_i \bmod 2^{m-1}\right) + \alpha_{i-1}$
...
$= 2\left(a_i \bmod 2^{m-1}\right) + \alpha_{i-1}$

Figure 7. A section Figure 6 showing the generation of the unknown carry bit c_{i+m}.

optimisation has improved the delay for squaring by a factor of 0.93.

Note that these comparisons are between multiplication using precomputed partial products and sliding windows and optimised squaring using precomputed partial products and sliding windows. Unfortunately, this for this combination of enhancements the benefit of optimised squaring is less than the theoretical speed improvement of 0.5 and also less than the 0.85 improvement reported in [8]. This is due to constant cost factors such as loop overhead as well as the cost of dealing with the correction factors. Nonetheless, a speed-up is obtained.

It is possible to extend this work to use a signed sliding window conversion to the digit-set $a_i \in \pm\{0, 1, 3, \ldots, 2^m - 1\}$. In this case only the positive precomputed multiples need be stored with negative partial squares being subtracted from the partial result. The derivation of the algorithm follows the unsigned case presented here but must deal with an extra correction factor that results from the selection of negative digits. In this case the best compromise between evaluation time, precomputation time and memory consumption for 512-bit squaring time occurred with $m = 5$ — squaring took 723 µs compared with 780 µs for multiplication. This was sufficient for a full 1024-bit signature to be generated in 1.82 seconds using 1984 bytes of data RAM and 1024 bytes of data EEPROM.

I conclude that the combination of optimised squaring with precomputed partial products is useful for some platforms, particularly where a fast multiplier is not available and arithmetic must be performed by an iterative procedure of additions and shifts.

5. References

[1] Bruce Schneier. *Applied Cryptography.* Second Edition, John Wiley and Sons, 1996.

[2] Donald E. Knuth. *The Art of Computer Programming*. Volume 2, Seminumerical Algorithms, Third Edition, Addison-Wesley, 1997.

[3] W. Rankl and W. Effing. *Smart Card Handbook*. Second Edition, John Wiley and Sons, 2000.

[4] Tien Chi Chen, "A Binary Multiplication Scheme Based on Squaring", *IEEE Transactions on Computers,* vol. 20, 1971, pp. 678-680.

[5] B. J. Phillips and N. Burgess, "Signed sliding window algorithms for modulo multiplication", *Electronics Letters*, vol. 36, no. 23, November 2000, pp.1925-1927.

[6] B. J. Phillips and N. Burgess, "Optimised Squaring with Sliding Windows", *Proceedings of the 34th Asilomar Conference on Signals, Systems and Computers*, Monterey, California, October 2000.

[7] M. Shand and J. Vuillemin, "Fast Implementations of RSA Cryptography", *Proceedings 11th IEEE Symposium on Computer Arithmetic,* IEEE Computer Society Press, 1993, pp. 252-259.

[8] Jean-François Dhem. *Design of an efficient public-key cryptographic library for RISC-based smart cards.* PhD Thesis, Université Catholique de Louvain, May 1998.

Find A^2 using optimised squaring and sliding windows with $m = 3$:

Pre-compute Stage. A table of pre-computed multiples is evaluated and stored. The correction values $t_{\min}(j)$ can also be determined in advance of the square.

$A = A_{0,1} =$	0	0	0	1	0	0	1	1	1	1	0	0	1	1	0	0	1
$3A = A + 2A = A_{0,3} =$	0	0	1	1	1	0	1	1	0	1	1	0	0	1	0	1	1
$5A = 3A + 2A = A_{0,5} =$	0	1	1	0	0	0	1	0	1	1	1	1	1	1	1	0	1
$7A = 5A + 2A = A_{0,7} =$	1	0	0	0	1	0	1	0	1	0	0	1	0	1	1	1	1

Using Equation 8 for $t_{\min}(j)$ gives:

$t_{\min}(1) = 0,\ t_{\min}(3) = 1,\ t_{\min}(5) = 3,\ t_{\min}(7) = 6$

Evaluate Stage. Sliding windows with $m = 3$ are applied to determine the non-zero digits a_i. For each non-zero digit a partial square is evaluated by truncating one of the precomputed multiples and applying a correction. The accumulation of partial squares is shown below.

Apply sliding windows gives $i = 0$, $a_i = 1$, $A =$	0	0	0	1	0	0	1	1	1	1	0	0	1	1	$\boxed{0}$	$\boxed{0}$	$\boxed{1}$
$t = t_{\min}(1) + d_3 \oplus \alpha_3 \oplus (t_{\min}(1) \bmod 2) = 0 + 1 \oplus 1 \oplus 0 = 0$																	
Find $A_{3,1} = (A_{0,1} >> 3) - t =$							1	0	0	1	1	1	1	0	0	1	1
Apply sliding windows gives $i = 3$, $a_i = 3$, $A =$	0	0	0	1	0	0	1	1	1	1	0	$\boxed{0}$	$\boxed{0}$	$\boxed{3}$	0	0	1
$t = t_{\min}(3) + d_6 \oplus \alpha_6 \oplus (t_{\min}(3) \bmod 2) = 1 + 1 \oplus 0 \oplus 1 = 1$																	
Find $A_{6,3} = (A_{0,3} >> 6) - t =$									1	1	1	0	1	1	0	1	0
Apply sliding windows gives $i = 7$, $a_i = 7$, $A =$	0	0	0	1	0	0	1	$\boxed{0}$	$\boxed{0}$	$\boxed{7}$	0	0	0	3	0	0	1
$t = t_{\min}(7) + d_{10} \oplus \alpha_{10} \oplus (t_{\min}(7) \bmod 2) = 6 + 1 \oplus 1 \oplus 0 = 6$																	
Find $A_{10,7} = (A_{0,7} >> 10) - t =$												1	1	1	1	1	1
Apply sliding windows gives $i = 10$, $a_i = 1$, $A =$	0	0	0	1	$\boxed{0}$	$\boxed{0}$	$\boxed{1}$	0	0	7	0	0	0	3	0	0	1
$t = t_{\min}(1) + d_{13} \oplus \alpha_{13} \oplus (t_{\min}(1) \bmod 2) = 0 + 1 \oplus 1 \oplus 0 = 0$																	
Find $A_{13,1} = (A_{0,1} >> 13) - t =$																	1
Apply sliding windows gives $i = 16$, $a_i = 1$, $A =$	0	$\boxed{0}$	$\boxed{0}$	$\boxed{1}$	0	0	1	0	0	7	0	0	0	3	0	0	1
$t = t_{\min}(1) + d_{16} \oplus \alpha_{16} \oplus (t_{\min}(1) \bmod 2) = 0 + 0 \oplus 0 \oplus 0 = 0$																	
Find $A_{16,1} = (A_{0,1} >> 16) - t =$																	0

Accumulation. The partial squares and digit squares (a_i^2) are accumulated to determine the final result.

$a_0^2 2^0$																											1
$+ A_{3,1} 2^4$													1	0	0	1	1	1	1	0	0	1	1				
$+ a_3^2 2^6$																		1	0	0	1						
$+ A_{6,3} 2^{10}$									1	1	1	0	1	1	0	1	0										
$+ a_7^2 2^{14}$								1	1	0	0	0	1														
$+ A_{10,7} 2^{18}$				1	1	1	1	1	1																		
$+ a_{10}^2 2^{20}$							1																				
$+ A_{13,1} 2^{24}$			1																								
$+ a_{13}^2 2^{26}$	1																										
$= A^2 =$	1	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1	1	1	0	0	0	1

Figure 8. An example of optimised squaring using precomputed partial products.