Efficient Software Implementations of Modular Exponentiation

Shay Gueron [1, 2]

[1] Department of Mathematics, University of Haifa, Israel
[2] Intel Architecture Group, Microprocessor and Chipset Development, Israel Development Center, Haifa, Israel

Abstract. RSA computations have a significant effect on the workloads of SSL/TLS servers, and therefore their software implementations on general purpose processors are an important target for optimization. We concentrate here on 512-bit modular exponentiation, used for 1024-bit RSA. We propose optimizations in two directions. At the primitives' level, we study and improve the performance of an "Almost" Montgomery Multiplication. At the exponentiation level, we propose a method to reduce the cost of protecting the w-ary exponentiation algorithm against cache/timing side channel attacks. Together, these lead to an efficient software implementation of 512-bit modular exponentiation, which outperforms the currently fastest publicly available alternative. When measured on the latest x86-64 architecture, the 2nd Generation Intel® Core™ processor, our implementation is 43% faster than that of the current version of OpenSSL (1.0.0d).

Keywords: modular arithmetic, modular exponentiation, Montgomery multiplication, RSA.

1 Introduction

The need for end-to-end security in the internet constantly increases the world-wide number (and percentage) of SSL/TLS connections. As a result, the cryptographic algorithms that support such secure communications become a critical computational load for servers, and therefore an important target for optimization (see [6]). The performance of RSA is an important case, because RSA is part of the handshake of practically all TSL/SSL sessions. Currently, the majority of RSA computations use 1024-bit moduli. Their performance translate to the performance of 512-bit modular exponentiations, which are therefore our focus.

Our study is motivated by observing that: a) Modular exponentiation consists of sequences of modular multiplications (or equivalent computations); b) The emergence of cache/timing side channel attacks on general purpose platforms, requires the incorporation of appropriate mitigation. We therefore optimize in these directions.

To assess our optimizations, we compare against the public implementations OpenSSL [4] and publications [2] and [9], as follows. The OpenSSL library is a very widely used (open source) software implementation. The availability of its source code makes it easy to study, tweak and measure, and it is therefore an important comparison baseline. Furthermore, optimizations that can be integrated and adopted by the OpenSSL library have potential benefit to many platforms world-wide. Our second comparison baseline is the recent publication [2]. It proposes a new approach, resulting in what is claimed thereof to be “the world’s fastest modular exponentiation implementation on IA processors”, thus setting the bar for performance. A few months after the publication of [2], the related software implementation was publicly published in [9]. This is highly optimized (assembly) code has even better performance than reported in [2], and we therefore use it for our comparison.

We point out some necessary updates. Publication [2] reports an implementation that is 37% faster than that of OpenSSL version 0.9.8h (which was the current version at that time), when run on the previous generation 2010 Intel® Core™ processor (which was the latest x86-64 processor at that time). These baselines have changed, and to remain current, we extend out study to the latest (and much improved) OpenSSL version (1.0.0d) and to the latest 2nd Generation Intel® Core™ processor.

We start with developing a fast implementation of what we call Almost Montgomery Multiplication (AMM), a variant of the Montgomery Multiplication (MM). We compare it to the AMM algorithm proposed in [2] (mistakenly claimed to be an MM algorithm). Subsequently, we propose a method to reduce the cost of protecting the w-ary exponentiation against cache/timing side channel attacks.

Our study produces several practical results. With our new method for dispersing tables in memory, the side channel protected w-ary exponentiation (for 512-bit operands) is optimized for a window size of w=4, rather than at w=5 which is the commonly used value (e.g., in OpenSSL and in [2]/[9]). We show that a few very simple changes in OpenSSL (version 1.0.0d) speed up its 512-bit modular exponentiation on the latest 2nd Generation Intel® Core™ processor by 5.5%. Finally, our new optimized software implementation of 512-bit modular exponentiation is shown to be 3-4% faster than the fastest available implementation of [9]. It is also 40% faster than OpenSSL (1.0.0d), and due to its structure, can be used directly as an OpenSSL update. As such, it will be contributed to the open source community.

2 Preliminaries

We discuss RSA cryptosystem with a 2n-bit modulus size $N = P \times Q$, where P and Q are n-bit primes. We denote the 2n-bit private exponent by d. Decryption of (2n-bit) C requires 2n-bit modular exponentiation $C^d \bmod N$. To use the Chinese Remainder Theorem, $d_1 = d \bmod (P-1)$, $d_2 = d \bmod (P-1)$, and $Qinv = Q^{-1} \bmod P$ are pre-computed. Two n-bit modular exponentiations $M_1 = C^{d1} \bmod P$ and $M_2 = C^{d2} \bmod Q$, are computed (M_1, M_2, d_1, d_2 are n-bit integers), and are recombined by $C^d \bmod N = M_2+(Qinv \times (M_1-M_2) \bmod P) \times Q$. Thus, the computational cost of 2n-bit RSA decryption is well approximate as the cost of two n-bit modular exponentiations. In our context, we can assume that by construction (of the RSA keys), $2^{n-1} < P, Q < 2^n$.

2.1 Montgomery Multiplications

The Montgomery Multiplication is a well-known efficient technique for computing modular exponentiation. We explain it briefly (see [4], [5], [7], [1] for details).

Definition 1. Let m be some odd integer (modulus), a, b be two integers such that $0 \leq a, b < m$, and t be a positive integer (hereafter, all the variables are non-negative integers). The Montgomery Multiplication of a by b, modulo m, with respect to t, is defined by MM (a, b) = $a \times b \times 2^{-t}$ mod m. We say that t is the Montgomery parameter.

Using MM's for modular exponentiation is based on two observations. 1) for any two integers $0 \leq a, b < m$, we have $0 \leq$ MM (a, b) $< m$; 2) if $0 \leq a, b < m$, c2 = 2^{2t} mod m, a' = MM (a, c2), b' = MM (b, c2), u' = MM (a', b'), and u = MM (u', 1), then u = $a \times b$ mod m. Observation 1 ("stability") allows for using the output of one MM as an input to a subsequent MM. Note that modular exponentiation algorithms consist of sequences of modular multiplications, and that for a given modulus, the constant c2 = 2^{2t} mod m can be pre-computed. Therefore, a^x mod m (for $0 \leq a < m$ and some integer x) can be computed by: a) Mapping the base (a) to the Montgomery domain, a' = MM (a, c2) b) Using an exponentiation algorithm while replacing modular multiplications with MM's c) Mapping the result back to the residues domain, u = MM (u', 1).

MM's computations can use two steps: 1) T = $a \times b$ ($< m^2$) 2) a "Montgomery Reduction" to obtain $T \times 2^{-t}$ mod m. This can leverage the fact that computing a square is faster than general multiplication. This can speed up MM's with a = b, which occur often in our context. We call such computations "Montgomery Squaring" (MSQR).

2.2 A Montgomery Reduction lemma

Lemma 1. Consider positive integers s, n, r ≥ 0, and (odd) $m < 2^n$. Assume that $T < 2^{n+r+s}$. Define

$$F = F(T, m, s) = \frac{T + m \times \left(\left((T \bmod 2^s) \times \left((-m^{-1}) \bmod 2^s\right)\right) \bmod 2^s\right)}{2^s} \tag{1}$$

Then, F mod m = $T \times 2^{-s}$ mod m, and $F < 2^{n+r} + m$. If, in addition $T < m \times 2^{r+s}$, then $F < m \times 2^r + m < 2^{n+r} + m$. It is convenient to view Lemma 1 as an algorithm as in Fig. 1.

```
Algorithm 1: Computing F(T, m, s)
Input: m < $2^n$ (odd modulus), T < $2^{n+r+s}$ (n, s > 0, r ≥ 0)
Output: F, satisfying the post-conditions (below)
Pre-computed: k0 = $-m^{-1}$ mod $2^s$
Flow
   1. T1 = T mod $2^s$
   2. Y = (T1 × k0) mod $2^s$
   3. T2 = Y × m
   4. T3 = (T + T2)
   5. F = T3 / $2^s$
Return F
Post-condition: F mod m = T×$2^{-s}$ mod m, and F < $2^{n+r}$ + m
                    (if also T < m ×$2^{r+s}$, then F < m×$2^r$ + m < $2^{n+r}$ + m)
```

Fig. 1. A Montgomery Reduction lemma.

Proof. By taking Equation (1) modulo 2^s, and modulo m, we get F mod m = $T \times 2^{-s}$ mod m, and $F < T/2^s + m$. The inequalities follow immediately.

2.3 Modular Exponentiation Using Montgomery Multiplications

We show the w-ary exponentiation (as implemented by OpenSSL. See also [7]). The flow is independent of the exponent bits, thus resists some side channel attacks. Fig. 2 (left panel) shows the w-ary modular exponentiation, implemented by MM's.

```
Algorithm 2: w-ary exponentiation
using Montgomery Multiplications
(MM's)

Input: m (odd modulus), a < m, x
such that
x = x0 + x1 × $2^w$ + … + xk × $2^{kw}$,
where 0 ≤ x0,x1, …, xk ≤ $2^w$-1
(i.e., x is written as k+1
"digits" in radix $2^w$)
Parameters:  s, w
w is the window size; s is a
Montgomery parameter

Pre-computed: c2 = $2^{2s}$ mod m
Output: $a^x$ mod m
Flow:
   1. a' = MM (a, c2)
   2. m[0] = MM (c2, 1)
   3. m[1] = a'
   4. For i = 2, …, $2^w$-1
      4.1. m[i] = MM (m[i-1], a')
   End For
   5. Store m[0], …, m[$2^w$-1]
        in a table (A)
   6. Retrieve m[0] from table A
   7. h = m[0]
   8. For i = k, …, 0 do
        8.1. For j = 1, …, w
             8.1.1. h = MSQR (h)
        End For
     8.2. Retrieve m[xi] from A
     8.3. h = MM (h, m[xi])
   End For
   9. h = MM (h, 1)
Return h
```

```
Algorithm 3: w-ary exponentiation
using Almost Montgomery
Multiplications (AMM's)

Input: m (odd modulus), a < B, x
such that
x = x0 + x1 × $2^w$ + … + xk × $2^{kw}$,
where 0 ≤ x0,x1, …, xk ≤ $2^w$-1
(i.e., x is written as k+1
"digits" in radix $2^w$)
Parameters:  s, w
w is the window size; s is a
Montgomery parameter

Pre-computed: c2 = $2^{2s}$ mod m
Output: $a^x$ mod m
Flow:
   1. a' = AMM(a, c2)
   2. m[0] = AMM(c2, 1)
   3. m[1] = a'
   4. For i = 2, …, $2^w$-1
      4.1. m[i] = AMM (m[i-1], a')
   End For
   5. Store m[0], …, m[$2^w$-1]
        in a table (A)
   6. Retrieve m[0] from table A
   7. h = m[0]
   8. For i = k, …, 0 do
        8.1. For j = 1, …, w
             8.1.1. h = AMSQR (h)
        End For
     8.2. Retrieve m[xi] from A
     8.3. h = AMM(h, m[xi])
   End For
   9. h = AMM(h, 1)
   10. REDUCE h modulo m
Return h
```

Fig. 2. The w-ary exponentiation algorithm. Left panel: using MM's. Right panel: using AMM's. Boldface notations (right panel) indicate the differences between MM's and AMM's.

The computational cost of modular exponentiation with window size w, is

$$\approx (k+1) \times w \times \text{Cost (MSQR)} + (k+1+2^w) \times \text{Cost(MM)} + \text{Cost(Store/Retrieve)} \qquad (2)$$

where the relation between n and k is: if w | n, then k = n/w-1; else, k = floor (n/w).

We use a "Store/Retrieve" notation to describe the cost of using the table (A), and discuss the details below. For n=512, the choice w = 5 is considered optimal. It requires a table of 32 512-bit values, 515 MSQR's and 135 MM's. For comparison, w= 6 requires a table of 64 512-bit values, 516 MSQR's and 150 MM's, and w = 4, requires a table of 16 512-bit values, 512 MSQR's and 144 MM's.

Some improvements of w-ary modular exponentiation are detailed in Appendix A.

3 Computing MM's

Fig. 3 shows the Word-by-Word Montgomery Multiplication (WW-MM) algorithm.

```
Algorithm 4: Word-by-Word
Montgomery Multiplication (WW-MM)
Input: $m < 2^n$ (odd modulus),
       $0 \le a, b, < m$, $n=s\times k$
Output: $a\times b\times 2^{-s} \bmod m$
Pre-computed: $k0 = -m^{-1} \bmod 2^s$
Flow
   1. $T = a\times b$
   For i = 1 to k do
      2. $T1 = T \bmod 2^s$
      3. $Y = T1 \times k0 \bmod 2^s$
      4. $T2 = Y \times m$
      5. $T3 = (T + T2)$
      6. $T = T3 / 2^s$
   End For
   7. If $T \ge m$ then $X = T - m$;
         else $X = T$
Return X
```

```
Algorithm 5: Word-by-Word Almost
Montgomery Multiplication (WW-AMM)
Input: $m < 2^n$ (odd modulus),
       $0 \le a, b, < 2^n$, $n=s\times k$
Output: $a\times b\times 2^{-s} \bmod m$
Pre-computed: $k0 = -m^{-1} \bmod 2^s$
Flow
   1. $T = a\times b$
   For i = 1 to k do
      2. $T1 = T \bmod 2^s$
      3. $Y = T1 \times k0 \bmod 2^s$
      4. $T2 = Y \times m$
      5. $T3 = (T + T2)$
      6. $T = T3 / 2^s$
   End For
   7. If $T \ge 2^n$ then $X = T - m$;
         else $X = T$
Return X
Post-condition:
$X \bmod m = a\times b\times 2^{-n} \bmod m$, and $X < 2^n$
```

Fig. 3. Left panel: Word by Word Montgomery Multiplication (WW-MM). Right panel: Word by Word Almost Montgomery Multiplication (WW-AMM). Boldface notations (right panel) indicate the differences between MM's and AMM's.

Proof of correctness (WW-MM). Start with $T = a\times b < m^2 < m\times 2^n$. Apply Lemma 1, k times, using $r = (n - i\times s)$ in iteration i, for i =1, 2, …, k, and the updated value of T. After k iterations we get $T < 2m$. Step 7 reduces T modulo m.

Computational efficiency (WW-MM). Step 1 requires an n-bit multiplication. Step 3 requires the low half an s-bit multiplication. Step 4 requires n-bit by s-bit. In iteration I, Step 5 adds an (n+s)-bit number to an (2n-(i-1)s)-bit number. Step 7 requires (conditional) subtraction of an n-bit integer from an (n+1)-bit integer. For side channel protection, the subtraction (of either m or 0) is always performed.

The WW-MM algorithm is well suited for architectures with an s-bit multiplier and adder. Specifically, s=64 is a natural choice for the 64-bit architectures (x86-64) that we study. OpenSSL uses it, together with the w-ary exponentiation with w=5.

4 Almost Montgomery Multiplications

For our implementation, we use Almost Montgomery Multiplication (AMM), which is a variant of MM (this variant is mentioned in [10] as "incomplete" MM).

Definition 2: Let m be an odd integer, and let a, b, t be positive integers such that $0 \leq a, b < B$ for some B. The Almost Montgomery Multiplication (AMM) of a by b, with respect to m and t, is an integer U satisfying the conditions 1) $U \bmod m = a\times b\times 2^{-t} \bmod m$ and 2) $U < B$. Almost Montgomery Square (AMSQR) is AMM where a=b.

Proof of correctness (WW-AMM): we use $B = 2^n$ here. Start with $T = a \times b < 2^{2n}$. Apply Lemma 1, k time, with $r = (n - i\times s)$, in iteration i, for $i = 1, 2, \ldots, k$, and the updated value of T. After k iterations, $T < 2^n+m$. Step 7 guarantees that $X<2^n$.

Fig. 2 (right panel) shows the WW-AMM (with $B = 2^n$). The proof shows that AMM's have a "stability" property (like MM's), and can be similarly used for modular exponentiation. Fig. 3 (right panel) shows the w-ary exponentiation algorithm using AMM's/AMSQR's. The computation of an AMM is almost identical to that that of the MM, except for the conditional subtraction step (step 7 in Fig. 3, both panels). AMM's enjoy a simpler condition check, which gives them some computational advantage. The computational cost of the w-ary exponentiation, when using AMM's, is similar to Equation (2), with the obvious adaptation.

Remark 1. If we assume $2^{n-1} < m < 2^n$, then the output (X) of AMM can be fully reduced modulo m by a single (conditional) subtraction of m (because $2^n - m < m$). Obviously, such subtraction should be done once, after the exponentiation flow. From the proof of Lemma 1, we see that the computation h = AMM (h, 1) (Step 9 of Algorithm 3) outputs a value which is smaller than m+1. Since in our context the modulus m is a prime, and we assume that the exponentiation base is nonzero, it follows that the result (h) is already reduced modulo m. So, in fact, the final subtraction step (Step 10) can be ignored.

4.1 The Two Step Folding AMM (from Ref. [2])

Reference [1] describes an algorithm for computing a 512-bit AMM[1], extending the Montgomery-Svoboda method [1]. In the particular parameters setting of [2] the algorithm reduces the 1024-bit integer $T = a\times b$ to a 768 bits ("folding"), then to 640-bit bits (second folding), and follows with an Almost Montgomery Reduction to 512 bits. Proper corrections compensate for "carry" bits (overflows) chopped away during the reductions. Fig. 4 generalizes the algorithm of [2] in a general parameters setting.

Proof of correctness (TSF-AMM). Steps 3-5 reduce X from 8s bits to 6s bit, possibly modifying it modulo m (by $2^{6s} \bmod m$), and accumulate cf1 for appropriately correcting the final result, modulo m. Similarly, Steps 6-8 reduce X from 6s bits to 5s

[1] Ref. [1] mistakenly claims that the algorithm computes a 512 bit MM ($a\times b\times 2^{-128} \bmod m$), but, as shown here, it actually computes 512-bit AMM.

bit, possibly modifying it modulo m (by 2^{5s} mod m), and accumulate cf2 for appropriate correction. For Steps 9-13 we apply Lemma 1 with r=0 and $T < 2^{5s} = 2^{n+s}$ remaining with a number bounded by 2^n+m. Step 14 compensates for the chopped off carry bits (Steps 5.2, 8.2, and 12.2), obtaining a number which is congruent, modulo m, to $a \times b \times 2^{-s}$, and bounded by $X < 2^n+2m$. Steps 15 and 16 (assuming $2^{n-1} < m < 2^n$) assure that the output is smaller than 2^n, satisfying the required post conditions.

```
Algorithm 6: Two-Step Folding Almost Montgomery Multiplication (TSF-AMM)
             (generalization of 1to general parameters setting)
Input: m < $2^n$ (odd modulus), 0 ≤ a, b < $2^n$; n = 4s
Output: X satisfying the Post-conditions
Pre-computed:
k1 = $-m^{-1}$ mod $2^s$
M1 = $2^{6s}$ mod m; M2 = $2^{5s}$ mod m
Tab[0] = 0   Tab[1] = $2^{4s}$ mod m   Tab[2] = $2^{4s}$ mod m  Tab[3] = $2^{4s+1}$ mod m
Tab[4] = $2^{5s}$ mod m   Tab[5] = ($2^{5s}$ + $2^{4s}$) mod m  Tab[6] = ($2^{5s}$ + $2^{4s}$) mod m
Tab[7] = ($2^{5s}$ + $2^{4s+1}$) mod m
Flow
   1. X = a×b
   2. cf1 = 0; cf2 = 0; cf3 = 0;
   3. Xh = floor (X/$2^{6s}$);
      Xl = X mod $2^{6s}$
   4. X = Xh × M1 + Xl
   5. If (X ≥ $2^{6s}$)
        5.1. cf1 = 1
        5.2. X = X mod $2^{6s}$
   6. Xh = floor (X/$2^{5s}$);
      Xl = X mod $2^{5s}$
   7. X = Xh × M1 + Xl
   8. If (X ≥ $2^{5s}$)
        8.1. cf2 = 1
        8.2. X = X mod $2^{5s}$
   9. Xl = X mod $2^s$
   10. Q = (Xl × k1) mod $2^s$
   11. X = X + m×Q
   12. If (X ≥ $2^{5s}$)
        12.1. cf3 = 1
        12.2. X = X mod $2^{5s}$
   13. X = X/$2^s$
   14. X = X + Tab[cf1 × 4 +
                   cf2 × 2 + cf3]
   15. If (X ≥ $2^{4s}$)
        15.1. X = X - m
   16. If (X ≥ $2^{4s}$)
        16.1. X = X - m
Return X
Post-condition:
X mod m = a×b×$2^{-s}$ mod m, and X < $2^n$
```

Fig. 4. Two-Step Folding Almost Montgomery Multiplication (TSFAMM)

Computational efficiency (TSF-AMM). Step 1 requires a single multiplication of n-bit integers. Step 4 requires the multiplication of 4s-bit by 2s-bit integer and an addition of two 6s-bit integers. Step 7 requires the multiplication of 4s-bit by s-bit integer and an addition of two 5s-bit integers. Step 10 requires the low half of the product of two s-bit integers. Step 11 requires multiplication of 4s-bit by s-bit integer and an addition of two 5s-bit integers. Step 14 requires the addition of two 4s-bit numbers. Steps 15, 16 require the (conditional) subtraction of two 4s-bit integers.

Remark 2. Reference [2] asserts that the TSF-AMM computes $a \times b \times 2^{-128}$ mod m (n=512, s=128), but this is incorrect. One counterexample is $m = 2^{511} + 111$, $a = 2^{511} + 110$, $b = 2^{510} + 53$, where the output exceeds m. Fig. 4, is has the correct statement.

Remark 3. TSF-AMM implicitly assumes $2^{n-1} < m < 2^n$ (Steps 15-16). In this case, a single conditional subtraction of m suffices to reduce the output modulo m. Reference [2] uses two successive conditional subtractions after the exponentiation sequence, but the second one is redundant. By Remark 1, the first subtraction is also redundant.

4.2 Comparing the WW-AMM and the TSF-AMM AMM algorithms

Since we are interested in performance on x64 architectures, it is convenient to view the operands as "multi-precision" numbers with 64-bit digits and count single precision operations. This count is easily done based on the above text (Appendix B provides a detailed comparison). However, we found that such theoretical analysis does not necessarily (at least not directly) reflect on the performance of particular software implementations. The actual performance depends heavily on the way that the code is written and optimized (and also on the architecture that it is tested on). For example combining the multiplication and reduction steps into "multiply-add" operation reduces the overall number of addition operations. Therefore, comparison should be based on measuring the performance of fully optimized code, and on a given architecture. The results of such comparisons are reported in Section 6. Note also that the algorithms require a different number of pre-computed values (to be resident in the cache). WW-AMM requires only one 512-bit pre-computed value. TSF-AMM algorithm requires a table (TAB in Fig. 4) with eight 512-bit pre-computed values, plus three 512-bit constants M1, M2 and k1.

5 Optimizing the w-ary exponentiation's Store/Retrieve

Cache based side channel attacks are a recent threat to software implementations of cryptographic algorithms. Due to such vulnerabilities (e.g., [9]), modular exponentiation code need to be written in a way that its memory access patterns (at the granularity of a cache line) do not leak secret information. This requires a special method for storing (in memory) and retrieving values from table A (see Fig. 2).

For n=512 and w=5, the table holds $2^w=32$ values, each one of 512-bit. OpenSSL tackles the problem by storing the bytes of each such value at addresses spaced by 2^w bytes. Reading a 512-bit value from the scattered table involves 64 move operations to/from memory (all cache lines are accessed, thus dependency on the exponent bits is avoided). For platforms where the cache lines consist of 64 bytes (the more common case), this implementation supports window sizes of up to w=6 (if the cache lines consist of 32 bytes, the implementation supports window size of up to w=5).

Reference [2] proposes a useful optimization that is tailored to platforms with cache lines of 64 bytes and the choice w=5. The 32 values of the table are split into 16-bit "words", which are stored at addresses spaced by 2^{w+1} words (i.e., 2×2^w bytes). This way, each 512-bit value of the table has one word in each of the cache lines spanned by the table. Here, retrieving a value from the table involves only 32 move operations - half the number required by the OpenSSL implementation (albeit with (acceptable) loss of generality).

Based on measuring the high cost of the side channel store/retrieve protection, we optimize this method further. We choose a window size of w=4, obtaining a table of only $2^w=16$ 512-bit values. This allows for scattering these 16 values in 32-bit "dwords" with spacing of 2^w dwords (i.e., 4×2^w bytes). The choice w=4 requires 144 AMM's (9 more than with w=5). On the other hand, retrieving a value from the table requires only 16 move operations, which is half the number of moves involved with the method of [2] and a quarter of the number of moves use by the OpenSSL implementation. In addition, the reduced table size with w=4 saves 1024 bytes (sixteen cache lines) in the first level cache, compared to w=5. The description (code snippet) of our proposed Store/Retrieve method is illustrated in Fig. 3.

```
STORE    (scatter a table at a "dword" granularity)
RETRIEVE (gather from the scattered table, at a "dword" granularity)
*table is a pointer to 1024 bytes of allocated memory (64B aligned).
*const points to the 512-bit value to be stored or retrieved.
index(from 0 to 15) is the index of the value to be stored/retrieved.
```

```
void dword_scatter(
   uint32_t *table,
   uint32_t *const, int index)
{
 int i;
 for(i=0; i<16; i++)
 {
  table[i*16 + index] = const[i];
 }
}
```

```
void dword_gather(
   uint32_t *const,
   uint32_t *table, int index)
{
 int i;
 for(i=0; i<16; i++)
 {
  const[i] = table[i*16 + index];
 }
}
```

Fig. 5. Side channel protected Store/Retrieve at a granularity of a "dword", facilitated by choosing a window size w = 4 (where the table consists of 16 512-bit values).

6 Results

This section provides the performance results of our study. The measurements methodology is detailed in Appendix C.

AMM and AMSQR's: To measure the performance of the WW-AMM algorithm, we wrote a new optimized code implementation (code is due to Vlad Krasnov). To measure the performance of the TSF-AMM algorithm, we isolated the functions *mont_mul_a3b* and *sqr_reduce* from [11]. For comparison to OpenSSL, we isolated the function *BN_mod_mul_montgomery*. The performance of these primitives was measured, separately, for AMM and AMSQR (when possible; OpenSSL's *BN_mod_mul_montgomery* interleaves the computations and does not optimize for AMSQR). Table 1 summarizes the results on the two processor generations.

Table 1. The performance of the MM/AMM algorithms, measured on different processors.

	OpenSSL 1.0.0d (WW-MM)	TSF-AMM [11]	WW-AMM (this paper)
	CPU Cycles		
Processor	AMM		
Previous Generation Intel® Core™	924	710	623
2nd Generation Intel® Core™	594	453	423
	AMSQR		
Previous Generation Intel® Core™	N/A	588	571
2nd Generation Intel® Core™	N/A	401	368

Table 1 shows that our WW-AMM implementation is consistently the fastest one. On the previous generation 2010 Intel® Core™ processor it is 33% faster than the OpenSSL implementation, and 12% faster than the TSF-AMM of [11]. Obviously, AMSQR's are faster than AMM's. On the 2nd Generation Intel® Core™ processor, the performance of all three algorithms is significantly improved (by more than 30%), and our WW-AMM remains the fastest one. To explain the overall speedup, we point out that the 2nd Generation Intel® Core™ processor improved the performance of the 64-bit multiplication and add-with-carry (ADC) instructions, compared to the previous generation processor: the latency of the mul64 instruction is reduced from 9 cycles to 4 cycles; the latency of ADC (with immediate=0) was reduced from 2 cycles to 1 cycle, and the throughput of ADC (all variants) was doubled.

The effect of the Store/Retrieve optimization: Table 2 shows the speedup obtained by injecting a few simple changes in OpenSSL's side channel protected Store/Retrieve implementation (changes made in the file bn_exp.c). The left column OpenSSL with "constant time" option selected (which is the default configuration which applies the side channel protection). The middle column shows the result of using the Store/Retrieve technique suggested in [2]. The right column shows the effect of our optimization. Adding the optimizations to OpenSSL is straightforward, and the performance gain is apparent. Our optimization speeds up OpenSSL's 512-bit modular exponentiation by 4.8% and by 5.5% on the respective processors.

Table 2. OpenSSL's 512-bit modular exponentiation with integrated optimization for side channel mitigated Store/Retrieve.

	OpenSSL 1.0.0h Constant time 512-bit modular exponentiation		
	OpenSSL (w=5)	Optimization of [2] w=5 and table scattered at the granularity of a word.	This paper w = 4 and table scattered at the granularity of a "dword"
Processor	CPU Cycles		
Previous Generation Intel® Core™	675,000	648,715	642,481
2nd Generation Intel® Core™	435,400	421,422	411,270

Full 512-bit modular exponentiation: Table 3 shows the performance of 512-bit modular exponentiation, comparing four implementations. The two left implementations measure the *BN_mod_exp_mont* modular exponentiation function, with and without the "constant time" mitigation (with constant time flag the function calls the function *OpenSSL BN_mod_exp_mont_consttime*). In both cases, OpenSSL uses the WW-MM based w-ary exponentiation with the default choice w=5). The middle column measures the *mod_exp_512* function of [11], which is a very well-crafted optimized implementation of the TSF-AMS algorithm (using w=5). The rightmost column measures our optimized implementations of the WW-AMM algorithm, using the w-ary exponentiation with w=4.

Table 3. The performance of 512-bit modular exponentiation.

	512-bit modular exponentiation			
	OpenSSL 1.0.0d *non* constant time	OpenSSL 1.0.0d constant time	Exponentiation using TSF-AMM [11]	This paper Exponentiation using WW-AMM (optimized)
	CPU Cycles			
Previous Generation Intel® Core™	625,000	675,000	399,668	383,280
2nd Generation Intel® Core™	413,600	435,400	258,133	249,962

The differences between the “constant time” and “non-constant time” OpenSSL implementations illustrate cost of the side channel mitigation (5-7% degradation). As before, the 2nd Generation Intel® Core™ processor offers a significant performance boost (35% on the fastest implementation). Our optimized WW-AMM based modular exponentiation is the fastest implementation on both processors: ~43% faster than OpenSSL and 3-4% faster than the fastest know implementation of [11].

7 Conclusion

This paper studied efficient software implementations of modular exponentiation, on general purpose x64 architectures, focusing on 512-bit modular exponentiation.

At the level of the “primitives”, we studied two AMM algorithms and identified the WW-AMM as the preferred method (in several respects).

At the exponentiation level, we improved the side-channel protected gathering/scattering method. This led to the counterintuitive conclusion that window size w=4 is a more efficient choice than the standard choice w=5.

In the results section, we demonstrated the significant performance speedup (35% on the faster implementation) of 2nd Generation Intel® Core™ processor compared to the previous generation, and detailed some micro-architectural characteristics that explain the differences. We also showed that our optimized modular exponentiation implementation (using WW-AMM and w-ary exponentiation with w=4) outperforms the fastest publicly known alternative implementations.

To explain why OpenSSL turns out to be significantly slower than our optimized implementation, although both use similar primitives (WW-MM and WW-AMM), we suggest the following: a) there is no optimization for squaring, because the WW-MM function interleaves the multiplication and the reduction steps (which leads to a less efficient WW-MM implementation); b) the gathering/scattering strategy (for the “constant time” w-ary exponentiation) is not efficient (perhaps because it is designed to capture a general window size and modulus size, and not optimized for n=512).

We point out that our modular implementation can be further improved by writing all the code as a single function in assembly function (as in [11]), thus avoiding function call overheads (and additional overheads due to the OpenSSL code's structure such as NULL pointers checks, checks of the integrity of the BIGNUM structure, enforced in each function call). For the sake of code's simplicity, readability, and maintainability, we decided to not pursue such further optimizations at this stage, and settled with modular C code for the w-ary exponentiation function.

Finally, we report that an optimized 1024-bit modular exponentiation was written as well, and is also ~40% faster than that of OpenSSL (there is no publicly available implementation of the TSF-AMM based implementation to compare with).

On the practical side, we point out that our WW-AMM based 512-bit (and 1024-bit) modular exponentiation was written with exactly the same API used by OpenSSL. Therefore (as we tested), it can be seamlessly incorporated in OpenSSL, and generate an updated OpenSSL library offering the reported 43% speedup. We intend to contribute the optimized code to the open source (OpenSSL) community.

Acknowledgements. I thank Vlad Krasnov for his coding efforts. I thank Xiaozhu Kang, Vlad Krasnov, Mike Kounavis, Zeev Sperber, and Bob Valentine, for helpful discussions.

8 References

1. Brent, R., Zimmermann, P.: Modern Computer Arithmetic, Cambridge University Press (2010) (retrieved from http://www.loria.fr/~zimmerma/mca/pub226.html).
2. Gopal, V., Guilford, J., Ozturk, E., Feghali, W., Wolrich, G., Dixon, M.: Fast and Constant-Time Implementation of Modular Exponentiation. In: 28th International Symposium on Reliable Distributed Systems. Niagara Falls, New York, U.S.A (2009). www.cse.buffalo.edu/srds2009/escs2009_submission_Gopal.pdf
3. Intel® 64 and IA-32 Architectures Optimization Reference Manual (2011) http://www.intel.com/Assets/PDF/manual/248966.pdf
4. Koç, Ç.K., Walter, C.D.: Montgomery Arithmetic. In: Encyclopedia of Cryptography and Security, Henk van Tilborg (ed), Springer, 394-298 (2005).
5. Koc, Ç.K., Kaliski, B.S.: Analyzing and Comparing Montgomery Multiplication Algorithms. In: Micro, 16(3): 26-33 (1996) http://islab.oregonstate.edu/papers/j37acmon.pdf
6. Kounavis, M.E., Kang, X., Grewal, K., Eszenyi, M., Gueron, S., Durham, D.: Encrypting the internet. In: Proceedings of the ACM SIGCOMM 2010 conference on SIGCOMM http://portal.acm.org/citation.cfm?id=1851182.1851200
7. Menezes, A.J., van Oorschot P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, 5th printing (2001).
8. OpenSSL: The Open Source toolkit for SSL/TLS, http://www.openssl.org/
9. Percival C.: Cache missing for fun and profit. In: www.daemonology.net/papers/htt.pdf (2005).
10. Yanık, T., Savaş, E., Koc, Ç.K.: Incomplete reduction in modular arithmetic. In: IEEE Proc. Comput. Digital Tech., 149:46-52 (2002).
11. Ying, H.: #2175: [PATCH] Optimization for 1024 bit RSA on x86_64 platform, posted Feb. 20, 2010, http://rt.openssl.org/Ticket/Display.html?id=2175&user=guest&pass=guest

Appendix B: some shortcuts in the exponentiation algorithm

Several additional optimizations help reducing the number of AMM's AMSQR's involved with the exponentiation that is shown in Fig. 2. For the table generation, note that m[0]= 2^n-m (assuming $2^{n-1} < m < 2^n$) requiring on AMM. Also, entries with an even index can be obtained by an AMSQR (rather than AMM). This way, the table generation requires one subtraction, 2^{w-1} AMM's and 2^{w-1}-1 AMSQR's. The loop over the exponent windows can start from (k-1) to save w AMSQR's. Fig. 6 shows the revised algorithm.

```
Algorithm 7: Improved w-ary exponentiation using Almost Montgomery
Multiplications (AMM's)

Input: $2^{n-1} \le m < 2^n$ (odd modulus), a < m (nonzero)
x such that $x = x0 + x1 \times 2^w + \dots + xk \times 2^{kw}$,
where $0 \le x0, x1, \dots, xk \le 2^w-1$
(i.e., x is written as k+1 "digits" in radix $2^w$)
Parameters:  s, w
w is the window size; s is a Montgomery parameter
Pre-computed: $c2 = 2^{2s}$ mod m
Output: $a^x$ mod m

Flow:
   1. a' = AMM(a, c2)
   2. m[0] = $2^{512}$ - m
   3. m[1] = a'
   4. For i = 1, …, $2^{w-1}$-1
        4.1. m[i×2] = AMSQR(m[i])
        4.2. m[i×2 + 1] = AMM(m[i×2], a')
   End For
   5. Store m[0], …, m[$2^w$-1] in a table A
   6. Retrieve m[xk] from table A
   7. h = m[xk]
   8. For i = k-1, …, 0 do
        8.1. For j = 1, …, w
             8.1.1. h = AMSQR (h)
        End For
        8.2. Retrieve m[xi] from table A
        8.3. h = AMM(h, m[xi])
   End For
   9. h = AMM(h, 1)
   Return h
```

Fig. 6. Improved w-ary exponentiation algorithm using AMM's.

Appendix A: Comparing the AMM Algorithms

Since we are interested in performance on x64 architectures, it is convenient to view the operands as "multi-precision" numbers with 64-bit digits and count single precision operations. Table 4 lists the steps for both algorithms, while (roughly) placing side by side, common/equivalent steps (the count for addition of variable length operands were averaged). Note also that the TSF-AMM has overhead of fetching constants from the table (Tab in Fig. 4).

Table 4. Comparison between the two AMM algorithms WW-AMM and TFS-AMM

Comparing WW-AAM and TSF-AMM for 512-bit operands	
TSF-AMM Algorithm 6	WW-AMM Algorithm 5
1 × 512-bit by 512-bit multiplication	1 × 512-bit by 512-bit multiplication
	8 × 64-bit multiplications mod 2^{64}
1 × 256-bit by 512-bit multiplication	8 × 64-bit by 512-bit multiplications
2 × 128-bit by 512-bit multiplication	
	8 × ≈ 800-bit plus 512-bit additions
1 × 768-bit addition	
1 × 128-bit multiplication mod 2^{128}	
2 × 640-bit addition	
1 × 512-bit addition	
	1 × 512-bit subtraction (with a condition check)
2 × 512-bit subtraction (with a condition check)	

It is important to realize that this direct count is only an approximation and not an accurate comparison. For example, software optimization allows for combining the multiplication and reduction steps into an efficient series of multiply-add operation, reducing the overall number of addition operations (for both algorithms).

Remark 4. The first step (a×b) is common and can be equally optimized, for squaring or via using other multiplication algorithms (e.g., Karatsuba's method). However, for 512-bit operands and the architectures that we experimented on, we could not improve the performance by using the Karatsuba multiplication.

Remark 5. If code size and simplicity (and not only performance) is a consideration, the WW-AMM algorithm can be implemented with s = 512 (and one iteration). In this case, the implementation can use only one function (for 512-bit multiplication), which would be called three times. Such implementation is slower than the optimized one.

Appendix C: measurements methodology

The experiments were carried out on two processors: the previous generation 2010 Intel® Core™ processors (specifically, Intel Core® i5-750) and the latest 2nd Generation Intel® Core™ processor (specifically, Intel Core® i5-2500).

Run runs were carried out on a system where the Intel® Turbo Boost Technology, the Intel® Hyper-Threading Technology, and the Enhanced Intel Speedstep® Technology were disabled. The operating system was OpenSuse 64 bits.

Each measured function was isolated, run 25,000 times (warm-up), followed by 100,000 iterations that were clocked (using the RDTSC instruction) and averaged. To minimize the effect of background tasks running on the system, each such experiment was repeated five times, and the minimum result was recorded.

All the reported performance numbers were obtained with the same measurement methodology.