J Cryptogr Eng (2012) 2:31–43
DOI 10.1007/s13389-012-0031-5

REGULAR PAPER

# Efficient software implementations of modular exponentiation

**Shay Gueron**

Received: 3 January 2012 / Accepted: 19 March 2012 / Published online: 5 April 2012
© Springer-Verlag 2012

**Abstract** The significant cost of RSA computations affects the efficiency and responsiveness of SSL/TLS servers, and therefore software implementations of RSA are an important target for optimization. To this end, we study here efficient software implementations of modular exponentiation, which are also protected against software side channel analyses. We target superior performance for the ubiquitous ×86_64 architectures, used in most server platforms. The paper proposes optimizations in several directions: the Montgomery multiplications primitives, the w-ary modular exponentiation flow, and reduced cost of side channel mitigation. For a comparison baseline, we use the current OpenSSL version, 1.0.0e. Our implementation—called "RSAZ"—is more than 1.6 times faster than OpenSSL for both 1,024 and 2,048-bit keys, on the previous generation 2010 Intel® Core™ processors and on the 2nd generation Intel® Core™ processors. The RSAZ code was contributed to OpenSSL as a patch, and improvements proposed in an earlier version of this paper have already been incorporated into the future OpenSSL version.

**Keywords** Modular arithmetic · Modular exponentiation · Montgomery multiplication · RSA

## 1 Introduction

The need for end-to-end security in the Internet constantly increases the worldwide 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 [15]). The performance of RSA is an important case, because RSA is a part of the handshake of TSL/SSL sessions.

Practically, all RSA usages have 1,024-bit or 2,048-bit keys, where, RSA1024 are currently the majority, and are the main optimization focus. However, the fraction of usages employing RSA2048 is sharply growing. Coupled with NIST's recommendation for key-lengths [4], RSA2048 becomes an important optimization target as well. The performance of RSA1024 translates directly to the performance of 512-bit modular exponentiations, and RSA2048 computations translate to 1,024-bit modular exponentiations. These computations are our focus.

Our goal is to study efficient software implementations of modular exponentiation, on general purpose ×86_64 architectures (x64 for short). Since today, cryptographic codes are also required to be resistant to threats stemming from different types of side channel analyses, we address this requirement throughout the study. We start the investigation by trying to optimize the relevant "primitives" of modular exponentiations, namely modular multiplications (or equivalents), and continue through optimizing the exponentiation flow itself. At the same time, we also focus on making sure that the implementation resists side channel analyses in a broad way, while reducing the cost of the necessary mitigations.

To assess our results, we compare the performance of our implementation against the public implementations OpenSSL [18] and publications [7,27], 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

S. Gueron (✉)
Department of Mathematics, University of Haifa, Haifa, Israel
e-mail: shay@math.haifa.ac.il

S. Gueron
Intel Corporation, Israel Development Center, Haifa, Israel

comparison baseline. Furthermore, optimizations that can be integrated and adopted by the OpenSSL library have potential benefit to many platforms worldwide. We relate to the currently latest OpenSSL version, 1.0.0e.

The second comparison baseline is the recent publication [7]. 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 a bar for performance. Shortly after [7] was published, the related software implementation was publicly posted [27], as an OpenSSL patch, in the form of an OpenSSL "engine" called "RSAX". This RSAX submission is a highly optimized (assembler) code for modular exponentiation. It is an updated version whose performance improves over the reported results of [7]. RSAX is wrapped as an engine, and not integrated into the main OpenSSL tree, because it is based on an algorithm that requires different and additional pre-computed constants (from what OpenSSL uses). Finally, RSAX implements only 512-bit modular exponentiation, hence supports only RSA1024.

We note that a secure modular exponentiation implementation consists of many ingredients that can be optimized in orthogonal directions. We provide here a step by step description of the various considerations in building the optimization, and explain the contribution of the different ingredients.

The result is a software implementation of modular exponentiation, which we call "RSAZ" (short for RSA ZARIZ—Hebrew for "quick"). RSAZ is compatible with the OpenSSL interface, and can be therefore integrated into its main path. For RSA1024, it outperforms RSAX. Compared to OpenSSL 1.0.0e, RSAZ1024 and RSAZ2048 show, respectively, a speedup factor of 1.72 and 1.61 on the previous generation 2010 Intel® Core™ processor, and a speedup factor of 1.62 and 1.64 on the 2nd generation Intel® Core™ processor. Per request from the OpenSSL Team, RSAZ has been released as a patch, and can be found in [8].

## 2 Preliminaries

We discuss RSA cryptosystem [16] 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 $C^d mod N$. To use the Chinese Remainder Theorem, $d_1 = d mod(P - 1)$, $d_2 = d mod(Q - 1)$, and $Qinv = Q^{-1} mod P$ are pre-computed. Two $n$-bit modular exponentiations $M_1 = C^{d1} mod P$ and $M_2 = C^{d2} mod Q$, are computed ($M_1, M_2, d_1, d_2$ are $n$-bit integers), and are recombined by $C^d mod N = M_2 + (Qinv \times (M_1 - M_2) mod P) \times Q$. 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), $2^{n-1} < P, Q < 2^n$.

## 3 Protecting software implementations against software side channels analysis

Software side channels analysis is a class of techniques that can be used for attacking cryptographic applications that run in a multi-tasking environment. Recent publications showed that when an unprivileged process ("Spy") runs in parallel to another ("Crypto") process, and some processor resources are shared (explicitly or implicitly), then Spy can extract information about the execution flow of Crypto. Two examples are the memory access patterns [20] and taken branches during executions [1,2,6]). Depending on the way that the crypto code operates, this side channel information can compromise the secrets (keys) of Crypto.

In our context of modular exponentiation, both the modulus and the exponent are secret. When the w-ary exponentiation [16] and variants of Montgomery multiplications are used (details in the subsequent sections), the knowledge of the following information may compromise the secrets:

1. Which of the Montgomery multiplications, computed during the exponentiation, require an "end reduction" step (for example, see [3,6,22,25]).
2. Which of the $2^w$ entries of the computed table are accessed during the exponentiation (for example, see [20]).

As a general concept, we say that a piece of code is "inherently protected against software side channel analysis" ("inherently protected" hereafter) in a given environment, if for any chosen input, volunteering the full details of the following items does not leak any sensitive information: (a) the addresses (at the granularity of a cache line) that were accesses (read/write); (b) the resolutions (taken/not taken) of the executed branches; (c) the executed instructions. We make our modular exponentiation implementation inherently protected.

## 4 Montgomery multiplications and almost Montgomery multiplications

Modular multiplication (or an equivalent) is a critical building block for modular exponentiations. This section explains the selection of our preferred algorithm.

### 4.1 Montgomery multiplications basics

The Montgomery multiplication [17] (MM hereafter) is a well-known efficient technique for computing modular exponentiation. Being already a classical algorithm, we explain it only briefly (more details can be found, e.g., in [5,13,14,16].

**Fig. 1** A Montgomery reduction lemma. Computing Montgomery multiplications

```
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)
```

**Definition 1** Let $m$ be a positive 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 $\mathrm{MM}(a, b) = a \times b \times 2^{-t} mod m$. We say that $2^t$ is the Montgomery parameter.

The use of MMs for modular exponentiation is based on two observations: (1) for any two integers $0 \leq a, b < m$, we have $0 \leq \mathrm{MM}\ (a, b) < m$; (2) if $0 \leq a, b < m$, $c2 = 2^{2t} mod m$, $a' = \mathrm{MM}(a, c2) b' = \mathrm{MM}(b, c2)$, $u' = \mathrm{MM}(a', b')$ and $u = \mathrm{MM}\ (u', 1)$, 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, $a' = \mathrm{MM}(a, c2)$; (b) using an exponentiation algorithm while replacing modular multiplications with MMs; (c) mapping the result back to the residues domain, $u = \mathrm{MM}(u', 1)$.

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, and speed up MMs with $a = b$, a case that occurs often in our context. We call such computations “Montgomery Squaring” (MSQR).

## 4.2 A Montgomery reduction lemma

We will use the following lemma to discuss several variants of Montgomery multiplications and equivalent constructions.

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

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

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.

*Proof* By taking Eq. 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.

Figure 2 shows the word-by-word Montgomery multiplication (WW-MM) algorithm.

*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 $i = 1, 2, \ldots, k$, 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 of an $s$-bit multiplication. Step 4 requires $n$-bit by $s$-bit multiplication. 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.

*Remark 1* The WW-MM algorithm is well suited for architectures with an $s$-bit multiplier and adder (i.e., an $s$-bit ALU). Specifically, $s = 64$ is a natural choice for the 64-bit architectures (x86-64) that we study.

*Remark 2* In the WW-MM algorithm (by Lemma 1), the value of $T$ after $k$ iterations (Steps 2–6) is $(a \times b + m \times (-m^{-1} \times a \times b \bmod 2^s))/2^s$. Thus, $T$ only satisfies the condition $T < 2m$, which means that it is not necessarily reduced modulo $m$ (there are examples where $T > m$). The conditional subtraction in Step 7 reduces $T$ modulo $m$, and is called the end reduction (ER) step. This step is known to be problematic from the side channel perspective (see details below).

## 4.3 Almost Montgomery multiplications

For our implementation, we use almost Montgomery multiplication (AMM), which is a variant of MM (this variant is mentioned in [26] as “incomplete” MM).

**Definition 2** Let $m$ be a positive odd integer, and let $a$, $b$, $t$ be positive integers such that $0 \leq a, b < B$ for some $B$.

**Fig. 2** *Left panel* word-by-word Montgomery multiplication (WW-MM). *Right panel* word-by-word almost Montgomery multiplication (WW-AMM). The concept of AMM is explained in Sect. 4.3, below. *Boldface* notations (*right panel*) indicate the differences between MMs and AMMs

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

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

The almost Montgomery multiplication (AMM) of $a$ by $b$, with respect to $m$ and $t$, is an integer $U$ satisfying the following conditions: (1) $U mod m = a \times b \times 2^{-t} mod m$; (2) $U < B$. Almost Montgomery square (AMSQR) is AMM where $a = b$.

Figure 4 (right panel) shows the WW-AMM algorithm (with $B = 2^n$). We prove the correctness of the algorithm below. The proof also shows that AMMs have a "stability" property (like MMs), and can therefore be used in a similar way, for modular exponentiation.

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

*Remark 3* AMM and MM are almost identical, where the only difference is in the ER step (Step 7 in Fig. 2, both panels). For MM, the condition (whether the result is $T$ or $X = T - m$) can be determined only by knowing the sign of $X = T - m$. Thus, X needs to be computed first (at least up to a point where its sign is evident). By contrast, AMM enjoys a simpler condition check, based just on the carry-out bit of the last addition in step 6, with the further advantage that the value of this carry bit already determines whether the output is $T$ or $T - m$. This can simplify the computations, although side channel resistance requires that the subtraction is always performed. We point out that in an environment where side channel resistance is not required, AMM has the performance advantage of allowing to skip the subtraction when appropriate.

*Remark 4* 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 $h = \mathrm{AMM}(h, 1)$ (Step 9 of Algorithm 6 in Fig. 4) 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.

### 4.4 The two-step folding AMM (from [7])

Reference [7] describes an algorithm for computing a 512-bit AMM[1], extending the Montgomery–Svoboda method [5]. In the particular parameters setting of [7] the algorithm reduces the 1,024-bit integer $T = a \times b$ to a 768 bits ("folding"), then to 640 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. Figure 3 generalizes the algorithm of [7] to a general parameters setting, followed by a proof of correctness (missing from [7]).

*Proof of correctness (TSF-AMM)* Steps 3–5 reduce $X$ from 8 $s$ bits to 6 $s$, possibly modifying it modulo $m$ (by $2^{6s} mod m$), and accumulate *cf1* to correct the final result, modulo $m$. Similarly, Steps 6–8 reduce $X$ from 6 $s$ bits to 5 $s$, possibly modifying it modulo $m$ (by $2^{5s} mod m$), and accumulate *cf2* to correct the final result modulo $m$. 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 chopping off carry bits in Steps 5.2, 8.2, and 12.2, to obtain 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.

*Computational efficiency (TSF-AMM)* Step 1 requires a single multiplication of $n$-bit integers. Step 4 requires the

[1] Ref. [7] mistakenly claims that the algorithm computes a 512-bit MM ($a \times b \times 2^{-128} mod m$), but, as shown here, it actually computes 512-bit AMM.

**Fig. 3** Two-step folding almost Montgomery multiplication (TSFAMM)

```
Algorithm 4: Two-Step Folding Almost Montgomery Multiplication (TSF-AMM)
             (generalization of [7] to 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
```

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 5* Reference [7] 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 $m$. Figure 3, gives the correct bound.

*Remark 6* 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 [7] uses two successive conditional subtractions after the exponentiation sequence, but the second one is redundant. By Remark 4, the first subtraction is also redundant.

### 4.5 Comparing the WW-AMM and the TSF-AMM algorithms

Our first step in optimizing modular exponentiation is determining the preferred alternative between TSF-AMM and WW-AMM (or WW-MM). We start with a few general observations.

*Remark 7* The "Almost" MM algorithm assumes a lower bound on the modulus (in our case, $2^{n-1} < m < 2^n$). The classical MM algorithm does not require such assumption.

*Remark 8* Both TSF-AMM and WW-AMM are almost MMs, but their outputs are not equal (because they use a different Montgomery parameter).

*Remark 9* By construction, the TSF-AMM algorithm is inherently an AMM. On the other hand, WW-AMM can be easily turned into WW-MM (and vice versa).

*Remark 10* The step $(a \times b)$ is common to both algorithms.

*Remark 11* WW-AMM and TSF-AMM require a different number of pre-computed values (to be resident in the cache for a real implementation). For $n$-bit operands, WW-AMM requires only one $n$-bit and one $s$-bit ($s = 64$ in our context) pre-computed values. The TSF-AMM algorithm requires a table (table in Fig. 3) with eight $n$-bit pre-computed values, two $n$-bit constants $M1$, $M2$, and an $(n/4)$-bit value $k1$.

*Remark 12* If code size and simplicity is a consideration, then WW-AMM is obviously preferable over TSF-AMM. Furthermore, note that WW-AMM can be modified to use $s = n$ and $k = 1$ (i.e., one big "word" and then only one iteration). In this case, an implementation can use only one function, for computing an $n$-bit multiplication, which would be called three times. We found that such implementation yields a simple code, but is slower than our optimized WW-AMM.

Since we are interested in performance on $\times 64$ architectures, it is convenient to view the operands as "multi-precision" numbers with 64-bit digits and count single precision operations (additions and multiplications). Table 1 lists the steps for both algorithms, for 512-bit operands (for WW-AMM, we averaged the count for addition of variable length operands).

**Table 1** WW-AMM and TFS-AMM algorithms: a direct breakdown of the computations

| For 512-bit operands | |
|---|---|
| TSF-AMM | WW-AMM |
| 1 × 512-bit by 512-bit multiplication | 1 × 512-bit by 512-bit multiplication |
| 1 × 256-bit by 512-bit multiply-add | 8 × 64-bit multiplication mod $2^{64}$ |
| 2 × 128-bit by 512-bit multiply-add | 8 × 64-bit by 512-bit multiply-add |
| 1 × 128-bit multiplication $mod 2^{128}$ | 1 × 512-bit subtraction |
| 1 × 512-bit addition | |
| 2 × 512-bit subtraction | |

The straightforward count shows that TSF-AMM involves 131 single precision multiplications, whereas WW-AMM has 136. TSF-AMM involves about 417 single precision additions while WW-AMM involves around 392 (we approximate 3 single precision additions operations per 1 single-precision multiplication).

However, this naïve count does not necessarily (at least not directly) reflect on the resulting performance (for example, because fetching constants from the table (table in Fig. 3) in an inherently protected way is an additional overhead associated with TSF-AMM). In practice, the performance depends heavily on the code's efficiency and optimization, and also on the architecture that it runs on. For example, software optimization can fuse the multiplication and the reduction steps into an efficient sequence of multiply-add operations, thus reducing the overall number of addition operations (for both algorithms). Therefore, true comparison should be based on measuring the performance of fully optimized codes, on a given architecture. These results are reported in Sect. 8, and indicate that the WW-AMM is the faster alternative. Therefore, we selected the WW-AMM as the preferred algorithm.

## 5 Protecting AMMs and MMs against software side channel analyses

To make an implementation of WW-MM (or WW-AMM) inherently protected, we focus on the end reduction (ER) step (Step 7 in Fig. 2) of the algorithm. There are three considerations to guarantee. The obvious one is making the execution time independent of the ER step, and this implies that the subtraction ($X = T - m$) must always take place. The second consideration is making the implementation branch-free. Finally, it is also required to guarantee that the memory access patterns of the implementation leak no information about the ER step. We show here that achieving inherent protection can be subtle, by explaining why OpenSSL's WW-MM implementation is not inherently protected.

### 5.1 OpenSSL's WW-MM implementation is not inherently protected

OpenSSL uses the WW-MM algorithm, where the relevant function is *bn_mul_mont* (in *crypto/bn/asm/x86_64-mont.pl*). The side channel-protected implementation variant is selected, by default, via the "BN_FLG_CONSTTIME" flag. We review only the ER step. The following code snippet from *bn_mul_mont* shows how the borrow bit from the subtraction ($X = T - m$) is used for delivering the appropriate output of the algorithm.

```
  sbb \$0,%rax    # if there was carry out rax = 2^64-1
  and %rax,$ap    # in that case use ap pointer
  not %rax
  mov $rp,$np
  and %rax,$np    # otherwise use np = rp pointer
  lea -1($num),$j
  or  $np,$ap     # ap=borrow?tp:rp
.align 16
.Lcopy:            # copy or in-place refresh (np = rp)
  mov ($ap,$j,8),%rax
  mov %rax,($rp,$j,8)  # rp[i]=tp[i]
  mov $i,(%rsp,$j,8)   # zap temporary vector
  dec $j
  jge .Lcopy
```

In the ".Lcopy" loop (lines 8–13 above), the borrow bit is used for loading either a Qword ($T[i]$) of $T$ or a Qword ($X[i]$) of $X$ into register rax (line 9). Then, the content of rax is written into the location of $X[i]$, and zero is written into the location of $T[i]$ (lines10-11). This either refreshes $X$ and zeros $T$, or copies $T$ onto $X$ and zeros $T$.

This flow computes $X$ and $T$ unconditionally, and is branch free. However, this implementation is *not* inherently protected because knowledge of the read access pattern (i.e., the code reads either from $T$ or from $X$) exactly reveals the information that needs to be concealed.

We clarify that this observation *does not* imply that there is a practical vulnerability, or that the implementation is unprotected. Observe that the code writes, unconditionally, both $X$ and $T$ (it zeros $T$) inside the loop. Therefore, it can be argued that in order to exploit the borrow-dependent reads, the Spy needs to instrument itself with resolution less than

the ".Lcopy" loop turnaround, and the latency of RDTSC is too high for that [21]. However, the advantage of an inherently protected implementation is that such argumentation is not even needed. We therefore show here an inherently protected and efficient implementation of WW-MM/WW-AMM.

### 5.2 An inherently protected implementation of MM

For the ER step, we determine if the output is $T$ or $X = T - m$, according to the sign of $X$. We hold the value (fixed in our context) of $(-m)$ in a memory location $L1$. When the additions (Step 6; Fig. 2, left panel) finish, we store $T$ in memory location $L2$, while also adding (unconditionally) $T$ to that memory location $L1$. This places $T$ in $L2$, $X = T - m$ in $L1$, while the carry-out bit of the last addition indicates which location hold the desired output. We then load Qwords from $L1$ and $L2$, into registers *reg1* and *reg2*, and use the conditional move instruction *CMOVE reg1, reg2* to place the desired result in memory location $L1$. This is repeated for all the Qwords in locations $L1$ and $L2$. The following code snippet shows the proposed implementation (for 512-bit operands).

```
# rcx and rax hold zero
# carry flag holds carry out from the last adc
# rbx points to the result
# rsi points to -m
sbbq  %rcx, %rcx
movq  %r8, (%rbx)    # load the result into GPRs
movq  %r9, 8(%rbx)
. . .
movq  %r15, 56(%rbx)

addq  (%rsi) %r8     # add -m to the result in register
adcq  8(%rbp), %r9
. . .
adcq  56(%rbp), %r15
sbbq  %rax, %rax     # if there was carry out, then
                     # the addition was unneeded
orq %rax, %rcx

movq  (%rbx), %rax   # conditionally load the unmodified
cmove %rax, %r8      # result based on the carry flag
movq  8(%rbx), %rdx
cmove %rdx, %r9
. . .
movq  56(%rbx), %rdx
cmove %rdx, %r15

movq  %r8, (%rbx)    # store the correct result
movq  %r9, 8(%rbx)
. . .
movq  %r15, 56(%rbx)
```

### 5.3 An inherently protected implementation of AMM

We use the following property of AMM: the carry out bit from the last addition operation of Step 6 determines whether the output should be $T$ or $T - m$. We first set register *reg0* to

```
Algorithm 5: w-ary exponentiation
using Montgomery Multiplications
(MM's)

Input: m (odd modulus),
a < m < 2^n,
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:  w (window size)

Pre-computed: c2 = 2^2n 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

* "reduce" only; no multiply by 1.
```

```
Algorithm 6: w-ary exponentiation
using Almost Montgomery
Multiplications (AMM's)

Input: m (odd modulus),
a < B (=2^n),
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:  w (window size)

Pre-computed: c2 = 2^2n 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. 4** The $w$-ary exponentiation algorithm. *Left panel* using MMs. *Right panel* using AMMs. *Boldface notations* (*right panel*) indicate the differences between MMs and AMMs

**Fig. 5** Improved $w$-ary exponentiation algorithm using AMMs

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

Input: 2^(n-1) ≤ m < 2^n (odd modulus), a < m (nonzero)
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] = 2^n - 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

* step 9 only "reduces"; avoids unnecessarily multiplying h by 1
```

the value "$2^{64}$ minus carry out bit". Then, load a Qword of $T$ into register *reg1*, and a Qword of $m$ into register *reg2*, write [*reg2 AND reg0*] into *reg2*, subtract *reg2* from *reg1*, and move *reg1* to the memory location where we want the result. This subtracts, correctly, either a Qword of $m$ or a zero, from $T$. No branches are used, and the memory access pattern is independent of the carry out bit, thus achieving an inherently protected implementation. The following snippet demonstrates the implementation.

```
# rcx holds zero
# carry flag holds carry out from last adc operation
# rbx points to the result
# rsi points to the modulus

sbbq  %rcx, %rcx  # subtract the carry flag from rcx -
                  # if set: rcx = 1^64; else rcx = 0^64
movq   (%rsi),   %r8   # load the modulus
movq   8(%rsi),  %r9
. . .
movq   56(%rsi), %r15
andq   %rcx, %r8       # if no reduction needed
andq   %rcx, %r9       # this will nullify the modulus
. . .
andq   %rcx, %r15
subq   %r8,  (%rbx)    # subtract from the result,
sbbq   %r9,  8(%rbx)   # if no carry zero is subtracted
. . .
sbbq   %r15, 56(%rbx)
```

## 6 The w-ary modular exponentiation using MMs and AMMs

We use the $w$-ary exponentiation algorithm (see e.g., [16]). The advantage of this algorithm (e.g., compared to a sliding window exponentiation) is that the flow is branch free, and is independent of the secret exponent bits (although, as we discuss below, other security considerations are required).

### 6.1 OpenSSL's w-ary modular exponentiation

Figure 4 (left panel) shows the $w$-ary modular exponentiation using MMs, as it is implemented by OpenSSL (1.0.0e). The right panel shows a flow that uses AMMs.

The computational cost of $w$-ary exponentiation with window size $w$, is

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

where the relation between $n$, $k$, and $w$, is the following:

$$\text{if } w \text{ divides } n, \text{ then } k = n/w - 1; \text{ else, } k = floor(n/w) \quad (3)$$

We use a "Store/Retrieve" notation to describe the cost of accessing the table (A in Fig. 4), and relate to this cost below. Special Store and Retrieve are required, at additional cost, in order to make the implementation inherently protected.

For $n = 512$, the choice $w = 5$ is considered optimal. It requires a table of 32,512-bit values, 515 MSQRs and 135 MMs. For comparison, $w = 6$ requires a table of 64512-bit values, 516 MSQRs and 150 MMs, and $w =$

4, requires a table of 16,512-bit values, 512 MSQRs and 144 MMs.

### 6.2 Introducing shortcuts to the w-ary exponentiation

We describe a variation of the $w$-ary exponentiation, with optimizations that reduce the number of AMM's AMSQRs.

1. For the table generation, note that $m[0] = 2^n modm = 2^n - m$ because it is assumed that $2^{n-1} < m < 2^n$. This saves one AMM. We comment that even if $m$ is smaller (and therefore $2^n - m$ is not fully reduced modulo $m$), the exponentiation would give the correct results when using AMMs (and all MMs), as long as the appropriate value of $k$ is used.
2. Entries with an even index can be obtained by AMSQR instead of AMM. Thus, the table generation can use one subtraction, $2^{w-1}$ AMMs and $2^{w-1} - 1$ AMSQRs.
3. The loop over the exponent windows can start from $(k - 1)$, saving the amount of $w$ AMSQRs.

In total, the modified algorithm saves $w$ AMSQRs, 2 AMMs and replaces $2^{w-1}$ AMMs by the faster AMSQRs. Obviously, the same optimizations can be applied to a $w$-ary exponentiation based on MMs. Figure 5 shows the revised algorithm.

## 7 Inherently protected w-ary exponentiation with reduced cost

Cache based side channel attacks are a recent threat to software implementations of cryptographic algorithms. Due to such vulnerabilities (e.g., [20]), 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 cache) and retrieving values from Table A (see Fig. 4).

For $n = 512$ and $w = 5$, the table holds $2^w = 32$ values, each one of 512-bit. OpenSSL tackles the problem by scattering the bytes of each value at addresses spaced by $2^w$ bytes. Gathering a 512-bit value from the scattered table involves 64-move operations (but since all cache lines are accessed, implicit 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 [7] 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., $2 \times 2^w$ bytes). This way, each 512-bit value of the table has one word in each of the cache lines spanned by the table. 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 optimized 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 the 16 values in 32-bit "dwords" with spacing of $2^w$ dwords (i.e., $4 \times 2^w$ bytes). The choice $w = 4$ requires 144 AMMs (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 [7] and a quarter of the number of moves use by the OpenSSL implementation. In addition, the reduced table size with $w = 4$ saves 1,024 bytes (16 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. 6.

## 8 Results

This section provides the performance results of our study.

### 8.1 Cycles count: 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).

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. 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 reported cycles count performance numbers were obtained with the same measurement methodology.

### 8.2 Cycles count for 512-bit AMM/AMSQR and 512-bit modular exponentiation

*AMM and AMSQRs* For the performance of WW-AMM/AMSQR we measured our new optimized implementation. For the performance of TSF-AMM we isolated the *mont_mul_a3b* and *sqr_reduce* functions from [27]. For comparison to OpenSSL we isolated its WW-MM implemen-

**Fig. 6** Inherently protected Store/Retrieve (for 512-bit modular exponentiation) at a granularity of a "dword", facilitated by choosing a window size $w = 4$, where the table consists of 16,512-bit values

```
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];
 }
}
```

**Table 2** The performance of 512-bit MM/AMM algorithms, and 512-bit modular exponentiation, in CPU cycles, measured on different processors

| Processor | OpenSSL 1.0.0e (WW-MM) | TSF-AMM [27] | WW-AMM (this paper) |
|---|---|---|---|
| CPU cycles: 512-bit AMM | | | |
| Westmere[a] | 924 | 710 | 637 |
| Sandy Bridge[b] | 594 | 453 | 428 |
| CPU cycles: 512-bit AMSQR | | | |
| Westmere[a] | N/A | 588 | 550 |
| Sandy Bridge[b] | N/A | 401 | 342 |
| CPU cycles: 512-bit modular exponentiation | | | |
| Westmere[a] | 675,000 | 399,668 | 358,499 |
| Sandy Bridge[b] | 435,400 | 258,133 | 230,959 |

[a] Previous generation Intel® Core™ i7 CPU X 980 @ 3.33 GHz (a.k.a. "Westmere")
[b] 2nd generation Intel® Core™ i7-2600 K CPU @ 3.40 GHz (a.k.a. "Sandy Bridge")

**Table 3** OpenSSL's 512-bit modular exponentiation with different optimizations for inherently protected Store/Retrieve

| Processor | OpenSSL 1.0.0e Constant time 512-bit modular exponentiation | | |
|---|---|---|---|
| | OpenSSL[c] | Optimization of [7][d] | This paper[f] |
| CPU cycles: 512-bit modular exponentiation | | | |
| Westmere[a] | 675,000 (625,000)[d] | 648,715 | 642,481 |
| Sandy Bridge[b] | 435,400 (413,600)[d] | 421,422 | 411,270 |

[a] Previous generation Intel® Core™ i7 CPU X 980 @ 3.33 GHz (a.k.a. "Westmere")
[b] 2nd generation Intel® Core™ i7-2600 K CPU @ 3.40 GHz (a.k.a. "Sandy Bridge")
[c] OpenSSL uses $w = 5$ and a table scattered at the granularity of a byte
[d] Numbers in parentheses are for OpenSSL un-mitigated (non constant-time) implementation
[e] [7] uses $w = 5$ and a table scattered at the granularity of a (16-bit) word
[f] Our optimization uses $w = 5$ and a table scattered at the granularity of a (32-bit) dword

tation in the *BN_mod_mul_montgomery* function. The performance of these primitives was measured, separately, for AMM and AMSQR (when possible; OpenSSLs *BN_mod_mul_montgomery* interleaves the computations and does not optimize for MSQR).

*512-bit modular exponentiation* We report the performance of 512-bit modular exponentiation, comparing four implementations. For OpenSSL, we measured the *BN_mod_exp_mont* 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 a WW-MM based $w$-ary exponentiation with the (default) window size $w = 5$. We also measured the *mod_exp_512* function of [27], which is a hand crafted optimized implementation using TSF-AMS and a $w$-ary exponentiation with $w = 5$. For our implementation, we report our optimized implementations of the WW-AMM

algorithm, using the $w$-ary exponentiation with $w = 4$. Table 2 summarizes the results for 512-bit operands. It demonstrates that our AMM/AMSQR implementation is consistently the fastest alternative. For example, on the previous generation 2010 Intel® Core™ processor it is 1.45 times faster than the OpenSSL implementation, and 1.11 times faster than the TSF-AMM of [27]. Obviously, AMSQRs are faster than AMMs. On the 2nd generation Intel® Core™ processor, all three implementations improve by more than *30 %*.

*The effect of the Store/Retrieve optimization* The results in Table 3 demonstrate the benefit of our Store/Retrieve optimization. To isolate its effect, we compared the performance of 512-bit modular exponentiation with OpenSSL 1.0.0e, and with two variants obtained by applying the simple changes needed to modify the Store/Retrieve implementation (changes were made in the file bn_exp.c). Our optimization achieves the fastest results. The results of the un-protected modular exponentiation show the cost of the mitigations (∼5–7 % degradation). Note that a change in the Store/Retrieve operations, introduced straightforwardly into OpenSSL, achieves noticeable performance gain.

### 8.3 Cycles count for 1,024-bit MM/AMM and 1,024-bit modular exponentiation

Table 4 summarizes the results for 1, 024-bit operands. Since a 1, 024-bit TSF-AMM implementation is not publicly available, we compare our implementation only to that of OpenSSL 1.0.0e. As for the case of 512-bit operands, our implementation is significantly faster on both processors.

OpenSSL distribution has a built-in test utility, which includes speed tests for various functions. This speed test performs RSA signing (=RSA decryption) operation for 10 seconds and report the average number of "RSA signs per second". Unlike the cycles count results, reported above, these performance numbers also depend on the processor's frequency. These are practically "bottom line results" because they account for the two exponentiations, the CRT recombination, base blinding, and other overheads. They represent the number of new SSL/TLS connections per second that a server can accept. To measure the results, we integrated our RSAZ implementation into OpenSSL, obtained the faster version, and measured the performance using the "openssl speed rsa1024" and "openssl speed rsa2048".

The results are shown in Table 5. They demonstrate a significant speedup factor achieved by our implementation, and also illustrate the relative speedup across the processor generations.

Another result, which worth mentioning, is the following. We prepared and measured a version of RSAZ that uses the standard MM/MSQR instead of AMM/AMSQR, and the performances difference is less than 1 %.

In addition, we generated a version of OpenSSL that integrated only the AMM/AMSQR functions (called from OpenSSL's mont_mul() function), changed the windows size parameter to $w = 4$, and added optimized gather/scatter functions. These partial optimizations were sufficient to achieve a significant portion of the overall speedup. For example, for RSA1024, we obtained a speedup factor of 1.46, compared to OpenSSL 1.0.0e (counting 5,320 signs/s).

## 9 Discussion and analysis of the results

*Explaining the performance differences between the different processors* The reported results demonstrate that RSA computations are faster on the 2nd generation Intel® Core™ processor, than on the Previous generation Intel® Core™. This speedup is mainly due to the improved per-

**Table 4** The performance of 1,024-bit MM/AMM algorithms, and 1,024-bit modular exponentiation, in CPU cycles, measured on different processors

| Processor | OpenSSL 1.0.0e (WW-MM) | WW-AMM (this paper) |
|---|---|---|
| CPU cycles: 1,024-bit AMM | | |
| Westmere[a] | 3,220 | 2,483 |
| Sandy Bridge[a] | 2,184 | 1,889 |
| CPU cycles: 1,024-bit AMSQR | | |
| Westmere[a] | N/A | 2,074 |
| Sandy Bridge[b] | N/A | 1,389 |
| CPU cycles: 1,024-bit modular exponentiation | | |
| Westmere[a] | 4,413,906 | 2,624,316 |
| Sandy Bridge[b] | 2,833,200 | 1,823,342 |

[a] Previous generation Intel® Core™ Intel® Core™ i7 CPU X 980 @ 3.33 GHz (a.k.a. architecture codename "Westmere")
[b] Second generation Intel® Core™ Intel® Core ™ i7-2600 K CPU @ 3.40 GHz (a.k.a. architecture codename "Sandy Bridge")

**Table 5** The performance of RSA1024 and RSA2048, measured using the "openssl speed" utility

| Processor | OpenSSL 1.0.0e | RSAZ (this paper) | RSAX (engine) |
|---|---|---|---|
| RSA1024 sign/s | | | |
| Westmere (speedup factor: 1.72)[a] | 2,297 | 3,957 | 3,670 |
| Sandy Bridge (speedup factor: 1.62)[b] | 3,646 | 5,908 | 5,462 |
| RSA2048 sign/s | | | |
| Westmere (speedup factor: 1.61)[a] | 368 | 592 | N/A |
| Sandy Bridge (speedup factor: 1.64)[b] | 519 | 853 | N/A |

The performance is measured in RSA signs per second. Speedup is measured as the ratio (RSAZ sign per second)/(OpenSSL sign per second)
[a] Previous generation Intel® Core™ Intel® Core™ i7 CPU X 980 @ 3.33 GHz (a.k.a. architecture codename "Westmere")
[b] 2nd generation Intel® Core™ Intel® Core™ i7-2600 K CPU @ 3.40 GHz (a.k.a. architecture codename "Sandy Bridge")

formance of the 64-bit multiplication (*MUL*) and add-with-carry (*ADC*) instructions, as follows: the latency of *MUL*, in the 2nd generation Intel® Core™ processor, is reduced from 9 cycles (on the Previous generation processor) to 4 cycles, and the latency of *ADC* (with immediate $= 0$) is reduced from 2 cycles to 1 cycle, while its throughput doubled.

In addition, the 2nd generation Intel® Core™ processor has a new feature, called the "Decoded Instruction Cache" (see [12]), which allows it to cache decoded instructions and execute them faster when they are re-invoked. An algorithm can be optimized by making its code stay resident in this cache.

*Explaining why OpenSSL has a slower implementation* The reported results show that OpenSSL (1.0.0e) is significantly slower than the optimized RSAZ, although both implementations use, basically, similar primitives (WW-MM and WW-AMM). To explain these performance differences, we suggest the following main reasons: (a) OpenSSL has no optimization for squaring, because its WW-MM function interleaves the multiplication and the reduction steps. This 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 $n = 512$); (c) the big-number "multiply-add" implementation, and the $w$-ary exponentiation flow are more efficient in RSAZ (see details on big-number squaring in [9]); (d) the RSAZ implementation optimized for two specific (important) key sizes, namely $n = 512$ and $n = 1{,}024$, while OpenSSL's code is more general. In addition, we also point out that OpenSSL's implementation optimizes not only performance, but also code simplicity, readability, maintainability, portability, and generality. These incur some overheads, which an optimized implementation can avoid.

*The future OpenSSL version* We mention here that OpenSSL has already a "development branch" [19]. This version has integrated several improvements from RSAZ (based on an earlier version of this paper [10], and personal discussions with their developments team [21]), while adhering to the generality requirements, constraints, and the coding style of this library. This version will soon become official. This implementation gives up some portion of the obtainable speedup of RSAZ for the sake of generality and maintainability, as shown in Fig. 7.

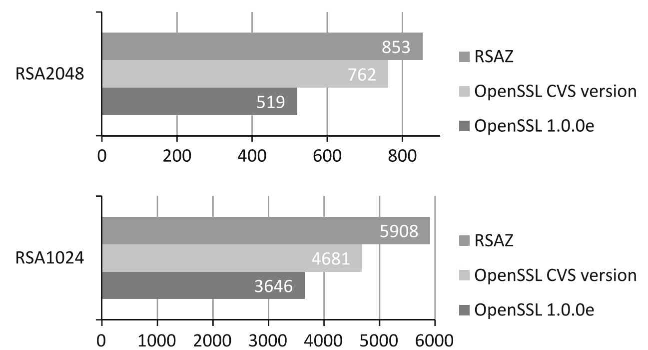


**Fig. 7** The performance of OpenSSL's development branch [19], versus OpenSSL 1.0.0e and RSAZ. The performance was measured using the 'openssl speed' utility on a *3.4 GHz* Intel® Core™ i7-2600 K CPU and reported in sign/sec

Finally, we point out that under some conditions the ER step in Montgomery multiplications can be simply skipped (see [11,23,25] for details). To use this property, the modulus needs to be shortened by two bits. In our context, this requires a change in the RSA primes generation (which OpenSSL generates), to satisfy $2^{n-1} < P, Q < 2^n$. With such shorter primes, removing the ER step makes the 512-bit modular exponentiation $\sim 4\,\%$ faster.

## 10 Conclusion

This paper studied efficient software implementations modular exponentiation, on general purpose $\times 64$ architectures, focusing on $512/1{,}024$-bit operands. It explained the algorithmic, software, and micro-architectural characteristics of the proposed optimizations, demonstrating their incremental contributions to the total speedup.

At the level of the "primitives", we identified the WW-AMM (and also WW-MM) method as the preferred algorithm for modular multiplication. At the exponentiation level, we reduced the cost of the associated side channel protection, and proposed a few improvements to the $w$-ary exponentiation. The combination of these, and an optimized code led to the new RSAZ implementation, offering a speedup factor of more than $1.6x$ over the current OpenSSL version (1.0.0e).

The RSAZ implementation can be seamlessly integrated into the OpenSSL library, as demonstrated in [8]. As per a request from the OpenSSL Development Team, the RSAZ code was contributed as a patch [8], for integration as a whole or in parts, for the benefit of the open source community. Some of the improvements, proposed in an earlier version of this paper, have already been incorporated into the next version of OpenSSL [19], achieving a gain factor of $1.54\times/1.49\times$ for RSA1024/RSA2048, respectively, on the Previous generation Intel Core Processors, and $1.28\times/1.47\times$ on the Second generation Intel Core Processors.

**Acknowledgments** I thank Vlad Krasnov for his coding and optimization efforts. I thank Ilya Albrekht, Xiaozhu Kang, Vlad Krasnov, Mike Kounavis, Max Locktyukhin, Ronny Ronen, Zeev Sperber, and Bob Valentine, for many helpful discussions. I also thank Andy Polyakov of the OpenSSL development team, for discussions and comments regarding the OpenSSL package, various considerations regarding software optimization and integration into OpenSSL, and comments on assembler implementations.

## References

1. Aciiçmez, O., Gueron, S., Seifert, J.P.: New Branch Prediction Vulnerabilities in Open SSL and necessary software countermeasures. In: Cryptography and Coding, 11th IMA International Conference (IMA Int. Conf. 2007), Lecture Notes in Computer Science, vol. 4887, pp. 185–203 (2007)
2. Aciiçmez, O., Kocç, C.K., Seifert, J.P.: Predicting secret keys via branch prediction. In: Lecture Notes in Computer Science, vol. 4377, pp. 225–242 (2007)
3. Aciiçmez, O., Schindler, W.: A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In: CT-RSA 2008, pp. 256–273 (2008)
4. Barker, E., Roginsky, A.: Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. In: NIST Special Publication 800-131A, p. 5 (2011). http://csrc.nist.gov/publications/nistpubs/800-131A/sp800-131A.pdf
5. Brent, R., Zimmermann, P.: Modern Computer Arithmetic. Cambridge University Press (2010). (retrieved from http://www.loria.fr/~zimmerma/mca/pub226.html)
6. Brumley, D., Boneh, D.: Remote timing attacks are practical. In: Proceedings of the 12th USENIX Security Symposium. pp. 1–14 (2003)
7. 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, USA (2009). http://www.cse.buffalo.edu/srds2009/escs2009_submission_Gopal.pdf
8. Gueron, S., Krasnov, V.: Efficient and side channel analysis resistant 512-bit and 1,024-bit modular exponentiation for optimizing RSA1024 and RSA2048 on ×86_64 platforms, OpenSSL #2582 patch. http://rt.openssl.org/Ticket/Display.html?id=2582&user=guest&pass=guest (posted Aug 2011)
9. Gueron, S., Krasnov, V.: Speeding up Big-Number Squaring. (to be published; ITNG 2012)
10. Gueron, S.: Efficient Software Implementations of Modular Exponentiation. eprint (2011) http://eprint.iacr.org/2011/239
11. Gueron, S.: Enhanced Montgomery multiplication. In: Cryptographic Hardware and Embedded Systems (CHES 2002), Lecture Notes in Computer Science, vol. 2523, pp. 46–56 (2002)
12. Intel: Intel® 64 and IA-32 Architectures Optimization Reference Manual (2011) http://www.intel.com/Assets/PDF/manual/248966.pdf
13. Koç, Ç.K., Walter, C.D.: Montgomery arithmetic. In: van Tilborg, H. (ed.) Encyclopedia of Cryptography and Security. pp. 298–394, Springer (2005)
14. Koc, Ç.K., Kaliski, B.S.: Analyzing and comparing Montgomery multiplication algorithms. Micro **16**(3), 26–33 (1996) http://islab.oregonstate.edu/papers/j37acmon.pdf
15. 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 (2010) http://portal.acm.org/citation.cfm?id=1851182.1851200
16. Menezes, A.J., van Oorschot P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, Boca Raton (2001) (5th printing)
17. Montgomery, P.L.: Modular Multiplication Without Trial Division. In: Mathematics of Computation, Volume 44:519–521 (1985)
18. OpenSSL: The Open Source toolkit for SSL/TLS. http://www.openssl.org/
19. OpenSSL: CVS Repository, http://cvs.openssl.org/
20. Percival C.: Cache missing for fun and profit. http://www.daemonology.net/papers/htt.pdf (2005)
21. Polyakov, A., OpenSSL Team (2011) (personal communications)
22. Schindler, W.: A timing attack against RSA with the Chinese Remainder Theorem. In: Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems, pp. 109–124. Springer (2000)
23. Walter, C.D.: Montgomery exponentiation needs no final subtractions. Electron. Lett. **35**, 1831–1832 (1999)
24. Walter, C.D.: An overview of Montgomery's multiplication technique: how to make it smaller and faster. In: Paar, C., Koç, Ç.K. (eds.) Cryptographic Hardware and Embedded Systems, Lecture Notes in Computer Science, vol. 1717, pp. 80–93 (1999)
25. Walter, C.D.: Precise Bounds for Montgomery Modular Multiplication and Some Potentially Insecure RSA Moduli. In: CT-RSA, vol. 2010, pp. 30–39 (2002)
26. Yanık, T., Savaş, E., Koc, Ç.K.: Incomplete reduction in modular arithmetic. IEEE Proc. Comput. Digital Tech. 149:46–52 (2002)
27. Ying, H.: Optimization for 1024 bit RSA on ×86_64 platform, OpenSSL #2175 patch, http://rt.openssl.org/Ticket/Display.html?id=2175&user=guest&pass=guest (posted Feb 20, 2010)