Fast and Constant-Time Implementation of Modular Exponentiation

Vinodh Gopal, James Guilford, Erdinc Ozturk, Wajdi Feghali, Gil Wolrich, Martin Dixon
Intel Corporation, USA
e-mail: vinodh.gopal@intel.com

***Abstract*— Modular exponentiation is an important operation which requires a vast amount of computations. Therefore, it is crucial to build fast exponentiation schemes. Since Cache and data-dependent branching behavior can alter the runtime of an algorithm significantly, it is also important to build an exponentiation scheme with constant run-time. However, such approaches have traditionally added significant overhead to the performance of the exponentiation computations due to costly mitigation steps. We present a novel constant run-time approach that results in the world's fastest modular exponentiation implementation on IA processors, bringing a 1.6X speedup to the fastest known modular exponentiation implementation in OpenSSL**

Keywords- modular, exponentiation, reduction, folding, Montgomery

I. INTRODUCTION

Modular exponentiation is an important operation which requires a vast amount of computations. Current fastest modular exponentiation algorithms are based on square-and-multiply method, which is described in Algorithm 1 derived from [5].

INPUT: g, p and a positive integer $e = (e_m e_{m-1} \ldots e_1 e_0)_2$.
OUTPUT: $g^e \bmod p$
1. A=1.
2. For i from m down to 0 do the following:
   2.1 $A=A^2 \bmod p$
   2.2 If $e_i = 1$, then $A=A*g \bmod p$
3. Return A

Algorithm 1: Left-to-right binary modular exponentiation with square-and-multiply method.

As can be seen from Algorithm 1, the building blocks of a modular exponentiation algorithm are modular squaring and modular multiplication operations. Therefore, efficient implementations of modular multiplication and modular squaring operations are highly important for modular exponentiation. Another method for optimizing modular exponentiation is reducing the number of modular multiplications for exponentiation calculations. For a square-and-multiply algorithm, the number of squaring operations is fixed and could not be reduced, so the windowing algorithms used focus on reducing the number of multiplications. Algorithm 2, derived from [5], shows the fixed-window algorithm used for modular exponentiation. For this algorithm, let n be defined as m/k and assume that m is divisble by k.

INPUT: g, p and a positive integer $e = (e_n e_{n-1} \ldots e_1 e_0)_b$, where $b = 2^k$ for some $k \geq 1$
OUTPUT: $g^e \bmod p$
1. *Precomputation*.
   1.1 $g_0=1$.
   1.2 For i from 1 to $(2^k - 1)$ do:
      1.2.1 $g_i=g_{i-1}*g \bmod p$
2. A=1.
3. For i from n down to 0 do:
   3.1 For j from 0 to k-1 do:
      3.1.1 $A = A^2 \bmod p$
   3.2 $A=A*g_{ei} \bmod p$
4. Return A

Algorithm 2: Left-to-right k-ary modular exponentiation with k-bit fixed windowing.

For Algorithm 1, there are m squaring operations and on average m/2 multiplications. For algorithm 2, there are n*k = m squaring operations and n multiplications. Since n=m/k, for k large enough to compensate the pre-computations, algorithm 2 becomes more efficient than Algorithm 1.

1.1 Modular Multiplication

There are two major methods that have been widely used for modular multiplication operations:

- Montgomery Multiplication-Reduction
- Full multiplication followed by a Barrett's reduction [4].

Traditional Montgomery approaches are combined multiply-reduce methods at the bit-level (mostly for hardware implementations) or word-level for software implementations (based on the processor's word-size). For Barrett's method, the two parts (multiply and reduce) can be optimized separately. Karatsuba algorithm [8] or classical multiplication algorithm could be utilized for a full multiplication operation.

Montgomery Multiplication could also be divided into two parts: a full multiplication followed by Montgomery Reduction. The Montgomery reduction will be defined on a large digit typically larger than the word-size for our approach. In this paper, we do not focus on the implementation of the full multiplication part and focus on the efficient reduction. We realize that by splitting the multiply from the reduction permits us to choose the best method for each resulting in the best overall performance.

1.1.1 Montgomery Reduction

Montgomery Reduction can be implemented in two ways: word-serial and bit-serial. For a software implementation, the bit-serial algorithm becomes too slow because the processor is built on word-level arithmetic. Therefore, software implementations typically utilize the word-level Montgomery Reduction algorithm. If we assume a word-level length of n, to reduce a 2n-bit number to an n-bit number, 2 full multiplications and 2 full addition operations are required. Thus, a full modular multiplication requires 3 multiplication and 2 addition operations [5]. This also applies to large digit approaches such as ours, where the multiplication/addition operations on large digits are further decomposed into word-size operations.

1.1.2 Barrett's Reduction

The Barrett reduction requires the pre-computation of one parameter, $\mu = \text{floor}(2^{2n}/M)$, where M is the modulus of the multiplication operation. Since this is a parameter that only depends on the modulus, it remains unchanged throughout the entire exponentiation operation, thus the calculation time of this parameter is not significant. If the number to be reduced is N, the reduction then takes the form

$$T1 = \mu * N$$
$$T2 = (T1/(2^n)) * M$$
$$R = (N-T2)/(2^n)$$

which requires two n-bit multiplies and one n-bit subtract, leaving the total at three multiplications and one subtraction. Clearly, R is congruent to N mod M, and it can be shown that $R < 3M$ [4].

1.1.3 Folding

In [3], the authors proposed a method called folding to reduce the amount of computations for a Barrett's reduction scheme. Their method relies on the precomputation of the constant $m' = 2^{3n/2} \bmod M$. The method is described in Figure 1.

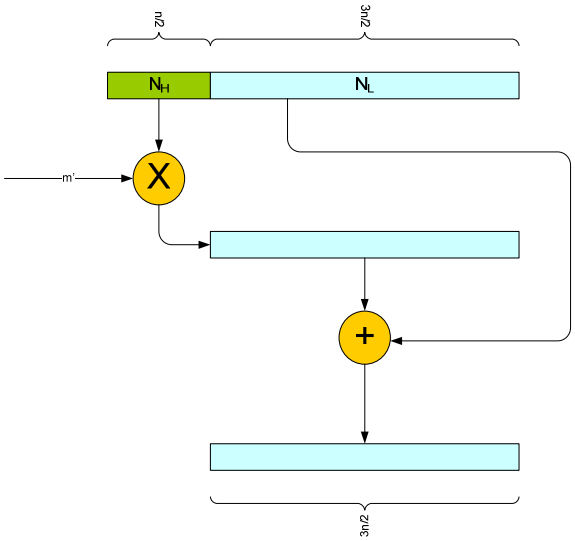

Figure 1: Folding method proposed in [3].

As can be seen in the figure, the highest n/2 bits of the number to be reduced is multiplied with $m' = 2^{3n/2} \bmod M$. Assume we call the highest n/2 bits of the number N as N_H and the rest N_L. The 2n-bit number N can be written as follows:

$$N = N_H * 2^{3n/2} + N_L$$

Now since we precomputed the value $2^{3n/2} \bmod M$, we can state that

$$N_H * 2^{3n/2} + N_L \equiv N_H * m' + N_L \bmod M$$

Since $N_H * m'$ is 3n/2 bits and N_L is 3n/2 bits, the resultant number is 3n/2 bits. Thus, this reduces the 2n-bit number N to 3n/2 bits. There can be a carry bit which increases the number of bits to 1+3n/2. This can be reduced by simply subtracting M from the resultant number. That extra bit could be also reduced in the further steps of the reduction algorithm. In a similar manner, one can perform another folding step and it was shown in [3] that the optimal number of folds is two.

1.2 Background on branch/cache access

The straightforward implementation of Algorithm 1 is not a constant run-time implementation, and the latency will be dependant on the data. A constant-time implementation is required. Instead of multiplying with g if the bit of the exponent is 1, we multiply with either 1 or g. This means that if the corresponding bit of the exponent is 0, we multiply with 1. If it is 1, we multiply with g. This results in a constant time binary exponentiation implementation (if the underlying modular multiply/square implementations are constant-time). For algorithm 2, the number of multiplications and squarings are fixed, thus the algorithm itself is constant-time, which means that it has same performance numbers for all different exponents. In addition, the implementation needs to be constant-time regarding to the cache accesses. The pre-computed values are stored in memory and for each multiplication; the corresponding location in the cache is accessed. Thus, it is also important to implement an exponentiation algorithm which has homogeneous cache-access mechanism for every exponent.

1.3 Motivation and prior work

In Barrett Reduction, there are two intermediate full multiplications to realize reduction. For both of these reductions, the high part of the previous result is used. For one of the multiplications, the low part is discarded. However, since the computation of the high part is dependent on the low part, both multiplications need to be fully computed. Thus, a full modular multiplication with Barrett Reduction method required 3 full multiplications. This is not the case for Montgomery Reduction. For Montgomery reduction, there are still 2 intermediate multiplications, but this time the low parts of the previous results are used for multiplications. Thus, one of the multiplications could actually be a half multiplication, because the high part of the result is discarded. Also, since the first intermediate multiplication is dependent only on the low part of the initial full multiplication, a pipelined architecture can better operate on the Montgomery Reduction, which means that the processor can

start computing the 1st intermediate multiplication before finishing the initial full multiplication. This makes Montgomery Reduction algorithm a better choice for software implementations.

Folding-based Barrett reduction is faster than the original Barrett Reduction. However, since each step produces extra bits and those bits need to be reduced before the final phase of the Barrett Reduction, a constant-time implementation will suffer from the constant time modulus subtractions. To resolve this issue, we combined the proposed folding method with Montgomery Reduction and our implementation has a superior performance over other implementations. It has the same structure but fewer odd-sized operands to deal with which results in a very effective method for making a time-invariant implementation. We then defined table sizes optimally to get cache dispersion for almost free. Also, for constant-time implementations, we picked a fixed-window approach.

The major improvement over the previous best techniques is to make a version of the reduction and modular exponentiation that has constant run-time. Mitigation techniques have been recently published that cause a significant loss in overall performance by striving to make the program constant-time (and thus equal its time for worst-case data) and by performing byte-level scatter-gather operations over cache-lines that defeats cache-based timing attacks [7].

II. High-level Overview

Our proposed scheme will be described in a simpler form that builds on the best-known Montgomery algorithm, modifies it with dual-folding and then suggests further improvements for constant-time mitigation and highest-performance.

III. Modified Montgomery reduction

Assume that we have a routine that can perform multi-precision multiply and add/subtract on 128-bit unsigned operands. Such a multiply routine can be implemented in software as a sequence of Classical/Karatsuba decompositions down to the smaller base multiplier size (likewise for the add/subtracts which can be easily implemented by a software routine as a sequence of add-with-carry operations). For convenience we also assume that such routines to add/subtract or multiply 256-bit numbers exist as well. We illustrate how to perform the reduction with 512-bit operands. It can be extended to any size by a natural extension.

Let a, b, m be 512 bit numbers. We require r = (a*b) mod m.

We compute X = a*b first separately using a mixed classical and 1-level Karatsuba bisection algorithm yielding a 1024 bit number. In what follows, we show an efficient way to reduce this number X w.r.t modulus m. Note that the modulus is required to be an odd number (a property required by any Montgomery scheme).

Montgomery Multiplication steps: compute $a*b*C^{-1} \bmod m$ (where $C = 2^{128}$)

```
X = a*b
cf=0;
Precompute M1 = 2^768 mod m
Xh = X >> 768
Xl = X % 2^768
X = Xh*M1 + Xl;  // 1st fold
If (X >= 2^768){
        Cf  |= 1;
        X = X % 2^768
}
cf <<=1;
M2 is a precomputed number = 2^640 mod m
Xh = X >> 640
Xl = X % 2^640
X = Xh*M2 + Xl;  // 2nd fold
If (X >= 2^640){
        Cf  |= 1;
        X = X % 2^640
}
cf <<=1;
```

Figure 2 shows these 2 folding steps in detail. At this point, we have done 2 folds and the number has been reduced from 1024 bits to 640 bits.

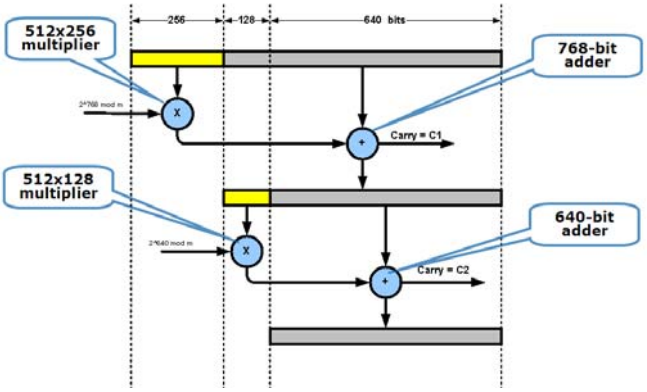

Figure 2: 2 folding steps for reduction

We save the intermediate carry-bits C1 and C2 (shown in Figure 2) in cf. Now, X has 128 more bits to be reduced to get a 512-bit residue.

Note that k1, a 128-bit number = $((-1)* m^{-1})$ mod 2^{128}; is precomputed as a non-negative number. This final reduction phase is shown in Figure 3 in detail.

```
Xl = X % 2^128;
Q = (Xl * k1) % 2^128;
X = X + m*Q;
If (X >= 2^640){
        Cf  |= 1;
        X = X % 2^640
}
```

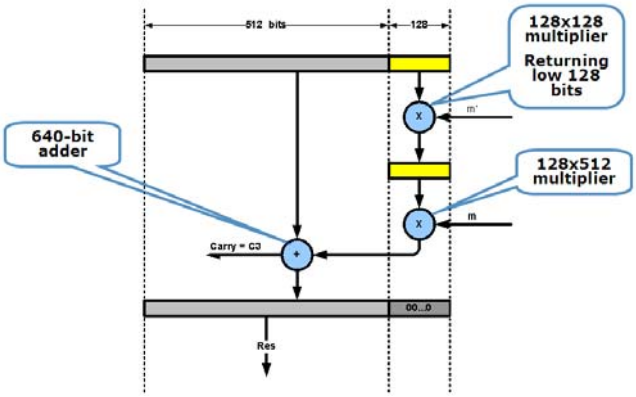

Figure 3: Final Montgomery reduction phase

Here the overflow bit C3 is stored in cf. Now we add correction based on value to be added to X using the overflow bits in cf. Note that X has 128 lsb's of 0

```
X = X >>128;
```

To implement a faster correction scheme, we precompute an adder table T:

000 : 0
001 : 2^{512} mod m
010 : 2^{512} mod m
011 : 2^{513} mod m
100 : 2^{640} mod m
101 : $2^{640} + 2^{512}$ mod m
110 : $2^{640} + 2^{512}$ mod m
111 : $2^{640} + 2^{513}$ mod m

The table is computed using the combinations of saved carry-bits. The msb was derived from the first save of 2^{768}, C1, followed by 2 bits that each saved 2^{640}, C2 and C3. Since we shifted X right by 128 bits, we use shifted exponents of 640 and 512 respectively.

Now we perform the corrections:

```
ShfAdd = T[cf];
X = X + ShfAdd;
If (X >= 2^512){        X = X – m; }
If (X >= 2^512){        X = X – m; }
Return (X);
```

Note that if the two condition IF statements above are implemented with a conditional branch, the program will not be branch-invariant in terms of timing; we thus need to make it constant time by always branching and subtracting m or 0 depending on the mask generated based on some value (513^{th} bit of X) – by making a mask we effectively turn the control dependency into a constant-time data dependency via the mask.

As another example consider:

```
If (X >= 2^640){
      Cf  |= 1;
      X = X % 2^640
}
```

to make this program constant-time, we cannot implement as shown above, because that would have different execution times depending on the data. Thus, to make it constant-time:

```
cf = cf | X[10];  // X as an array of 64-bit quadwords
```

Here we get the most-significant quadword X[10] which is either 0 or 1 and or it to cf, and then zero X[10]:

```
X[10] =  0;
```

Once we have a function montmul(A,B) which returns the value $A*B*C^{-1}$ mod m, (where $C = 2^{128}$) we show a simple exponentiation method (which can be extended to work with windowing schemes) for illustration. Assume n-bit operands. We compute:

```
X = montmul(x, (C^2 mod m));
A = C mod m;
for i = (n-1) downto 0 do {
    A = montmul(A, A);
    If (e_i == 1) A = montmul(A, X);
}
a = montmul(A, 1);
if (a >= m) a = a - m;
if (a >= m) a = a - m;
return a;
```

Here we implement the if (e_i == 1) statement with unconditional multiplication using masked X (X or C) to make the implementation constant-time.

IV. Methods to perform efficient folding with constant-time table lookups

Here is an efficient method that performs constant-time cache accesses for 5-bit fixed windows (which consists of 2^5 or 32 vectors). For 512-bit operand sizes, one can view the operands as 64 bytes of data. Rather than use a byte-level scatter/gather operation over cache lines, we use the fact that the IA (x86) cache line size is 64 bytes and we need 32 vectors to be stored, to implement a better scheme. The entire table must be aligned to start on a cache-line.

Let us view each vector as an array of unsigned short int (2 bytes each). Thus a vector is an array of 32 shorts. If the base pointer to the table is of type short and represented by bptr, then these are the steps to access a vector with a 5-bit index, "index":

```
short * start = bptr[index]; // 0 <= index <32
vec[0] = *start;
for (i=1; i<32; i++){
    start = start + 32; // same position in next cache line, 64 bytes away
    vec[i] = *start;
}
```

Now we can read the vec[] array as a series of quadwords. If we use the same technique to create (write) the tables as we do to read the vectors back, a simple scheme will work for accessing the quadwords without any byte-shuffling operations, as for example with:

```
Quadword0 = *( (uint64* ) vec);
```

Note that the 1st precomputed table in the modified Montgomery algorithm has only 8 vectors which is trivial to implement across cache-lines. We just treat each vector as an array of 8 quadwords which are distributed one per cache line. This works well because we have exactly 8 vectors occupying 8 cache lines and each cache line holds 8 quadwords.

V. RESULTS

Using this method we can design very high-speed modular exponentiation algorithms that have constant run-times. Current methods do not achieve such high performance.

Table 1: Performance results of 512-bit modular exponentiation when integrated in OpenSSL between our implementation and OpenSSL version 0.9.8h on Nehalem hardware in cycles measured with rdtsc.

	Latency (cycles)
OpenSSL Modular Exponentiation	704090
Our Modular Exponentiation	446669
Performance gain	704090/446669 = ~1.6X

As can be seen in Table 1, our proposed method results in **1.6X speedup of OpenSSL** which is a key benchmark for Servers.

The test was run on a 2.93 GHz Nehalem platform with 64-bit Linux operating system. Similar performance gains are also observed on other Intel platforms such as those based on the Merom/Woodcrest platforms.

SUMMARY

This method is very novel in terms of extending one of the best-known reduction methods (Montgomery Reduction) with folding.

By using table lookups combined with carry-saving techniques, we get a perfect implementation of modified Montgomery that is also time-invariant and as efficient as the unmitigated version. Our cache dispersion techniques are optimal and ensure that we perform large word level dispersions to get the same effect as slower byte scatter/gather schemes. We also demonstrate how this can be combined with exponent windowing and yet maintain constant-time invariance.

REFERENCES

[1] OpenSSL: The Open Source toolkit for SSL/TLS, http://www.openssl.org/

[2] Ronald L. Rivest, Adi Shamir, Leonard M. Adleman, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems” Commun. ACM 21(2): 120-126 (1978)

[3] William Hasenplaugh, Gunnar Gaubatz, Vinodh Gopal, "Fast Modular Reduction" 18th IEEE Symposium on Computer Arithmetic (ARITH '07), 2007, pp.225-229

[4] Paul Barrett. “Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor” in A. M. Odlyzko, editor, Advances in Cryptology, CRYPTO’86, volume 263 of Lecture Notes in Computer Science, pages 311–326, Heidelberg, Jan 1987. Springer-Verlag.

[5] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone: Handbook of Applied Cryptography, CRC Press, 1996.

[6] P. Kocher, Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS and Other Systems," In Advances in Cryptology, CRYPTO'96, LNCS 1109, pp. 104-113 Springer-Verlag, 1996.

[7] Brumley, D. and Boneh, D. 2003. Remote timing attacks are practical. In Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (Washington, DC, August 04 - 08, 2003). USENIX Security Symposium. USENIX Association, Berkeley, CA.

[8] A. Karatsuba and Y. Ofman. “Multiplication of multidigit numbers by automata” Soviet Physics-Doklady, 7:595–596, 1963.