Technology Guide

# Intel® AVX-512 - Fast Modular Multiplication Technique


## Authors

Erdinc Ozturk

Tomasz Kantecki

Kirk Yap


## 1 Introduction

The most commonly utilized public key cryptosystems (RSA, ECDSA, etc.) in the pre-quantum era are constructed over finite fields. For these finite fields, the most important and time-consuming operation is the modular multiplication of integers under large moduli. Efficient implementations of multi-precision arithmetic are very critical for the performance of these cryptosystems. Intel® Advanced Vector Extensions 512 (Intel® AVX-512) based Integer Fused Multiply Add Instructions (IFMA) are utilized for multi-buffer high-throughput software implementations of RSA. We present a novel modular multiplication algorithm that increases the throughput of multi-buffer IFMA implementations of RSA operations in the range of 10%.

RSA is the most popular digital signature algorithm used when establishing a connection using the Transport Layer Security (TLS) protocol. An improvement in its efficiency translates into better responsiveness of TLS or web servers.

This document is part of the Network Transformation Experience Kits.

# Table of Contents


1 Introduction ..... 1
1.1 Terminology ..... 3
1.2 Reference Documentation ..... 3

2 Overview ..... 3
2.1 Montgomery Reduction ..... 3
2.2 Montgomery Multiplication ..... 4
2.3 Technology Description ..... 5
2.3.1 Almost Half Montgomery Multiplication ..... 5

3 Performance Result ..... 7
3.1 Platform Configuration ..... 7
3.1.1 OpenSSL ..... 7
3.1.2 Intel® Integrated Performance Primitives Cryptography (Intel® IPP Cryptography) ..... 7
3.1.3 Intel Multi-Buffer Crypto for IPsec (intel-ipsec-mb) ..... 8
3.1.4 Intel QAT Engine ..... 8
3.2 OpenSSL Speed Data ..... 8
3.2.1 Baseline RSA2048 Version ..... 8
3.2.2 New RSA2048 Version ..... 9
3.2.3 OpenSSL Speed RSA2048 Summary ..... 9

4 Benefits ..... 9

5 Summary ..... 10

Appendix A System Configuration ..... 10


# Figures


Figure 1. Reduction Operation ..... 4
Figure 2. Single Iteration of the Reduction Loop as shown in Algorithm 1. ..... 4
Figure 3. Single Iteration of "Almost" Montgomery Multiplication Operation ..... 5
Figure 4. Almost Half Montgomery Multiplication ..... 6
Figure 5. RSA2048 Sign/s Performance Results (higher is better) ..... 9


# Tables


Table 1. Terminology ..... 3
Table 2. Reference Documents ..... 3
Table 3. Preliminaries ..... 3
Table 4. Algorithm 1 Word-level Almost Montgomery Reduction ..... 3
Table 5. Hardware Configuration ..... 10
Table 6. Software Configuration ..... 10


# Document Revision History

| Revision | Date | Description |
|---|---|---|
| 001 | March 2024 | Initial release. |

## 1.1 Terminology

Table 1. Terminology

| Abbreviation | Description |
|---|---|
| ECDSA | Elliptic Curve Digital Signature Algorithm |
| IFMA | Integer Fused Multiply Add Instructions |
| IKE | Internet Key Exchange |
| TLS | Transport Layer Security |

## 1.2 Reference Documentation

Table 2. Reference Documents

| Reference | Source |
|---|---|
| OpenSSL | https://github.com/openssl/openssl |
| Intel QAT Engine | https://github.com/intel/QAT_Engine |
| Intel IPP Crypto | https://github.com/intel/ipp-crypto |
| Intel Multi-Buffer Crypto for IPsec | https://github.com/intel/intel-ipsec-mb |
| Top 1 Million Analysis - November 2021 | https://scotthelme.co.uk/top-1-million-analysis-november-2021/ |

# 2 Overview

## 2.1 Montgomery Reduction

Modular Multiplication and Reduction operations can be defined and implemented in many ways. To demonstrate our technique, we utilize a word-level Montgomery Reduction approach. First, we define preliminaries as shown in Table 3.

Table 3. Preliminaries

| | |
|---|---|
| **k** | Bit-length of operands |
| **w** | word size in bits |
| **L** | number of words of operands = ⌈k/w⌉ |
| **M** | k-bit modulus |
| **R** | $2^{L*w}$ mod M |
| **mu** | $-M^{-1}$ mod $2^w$ (w-bit montgomery constant) |
| **A[i]** | $i^{th}$ word of A |

Montgomery Reduction can be utilized after performing full multiplication of the operands. This process is defined as follows:

Table 4. Algorithm 1 Word-level Almost Montgomery Reduction

| | |
|---|---|
| **Inputs:** | M, mu<br>C (2k-bit integer, C=A*B) |
| **Output:** | MR(C,M)=C*$R^{-1}$ mod M |
| **Operation:** | for i from 0 to (L-1):<br>T=C[0]*mu // discard T[1]<br>C=C+T[0]*M // C[0] is zero<br>C=C>>w // at each step of the for loop,<br>// divide the result by $2^w$. |

The entire reduction operation is depicted in Figure 1. A single iteration of the reduction loop is depicted in Figure 2. Note that this algorithm does not exactly represent Montgomery Reduction, but it represents an "Almost" Montgomery Reduction operation. In "Almost" Montgomery Reduction operation, the result is not less than the modulus M, but it fits into L words.

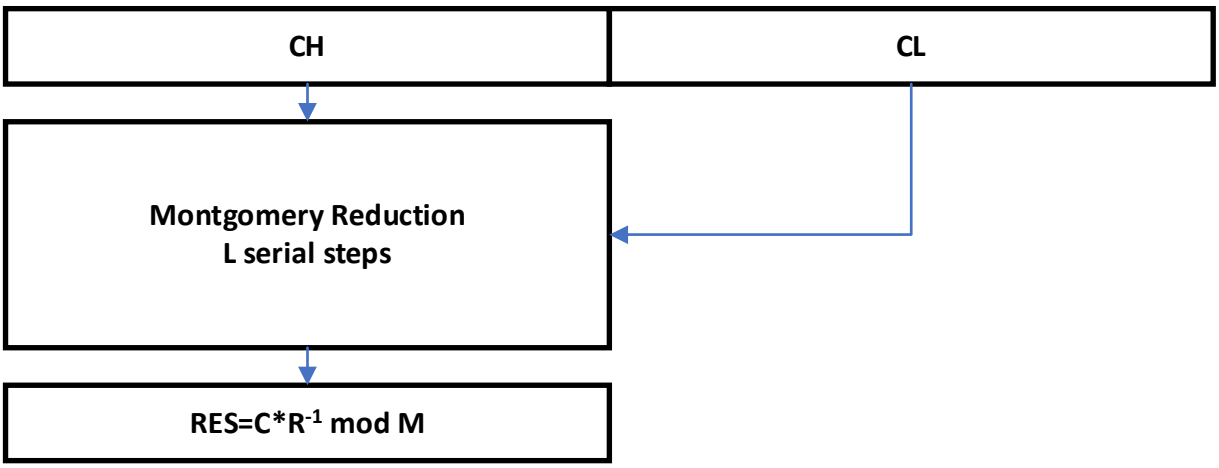


Figure 1. Reduction Operation

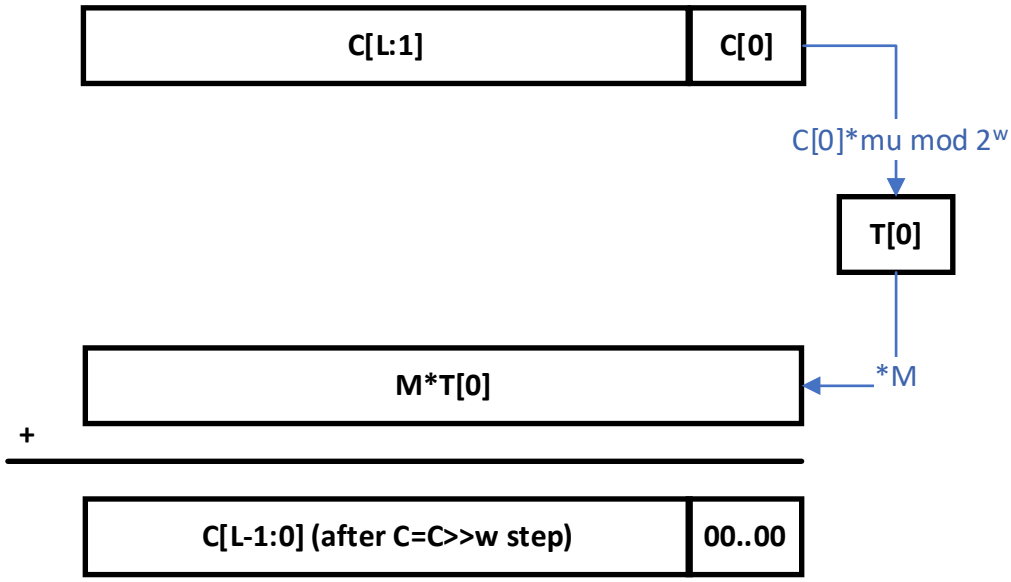


Figure 2. Single Iteration of the Reduction Loop as shown in Algorithm 1.

## 2.2 Montgomery Multiplication

The Montgomery Multiplication operation interleaves multiplication and reduction components. The "Almost" Montgomery Multiplication algorithm can be defined as follows:

| | |
|---|---|
| **Inputs:** | A,B (k-bit integers)<br>M, mu |
| **Output:** | MM(A,B,M) =A*B*R$^{-1}$ mod M |
| **Operation:** | C = 0<br>for i from 0 to (L-1):<br>C=C+A*B[i]<br>T=C[0]*mu // discard T[1]<br>C=C+T[0]*M // C[0] is zero<br>C=C>>w // at each step of the for loop,<br>// divide the result by 2$^{w}$. |

A single iteration of the "Almost" Montgomery Multiplication operation is depicted in Figure 3.

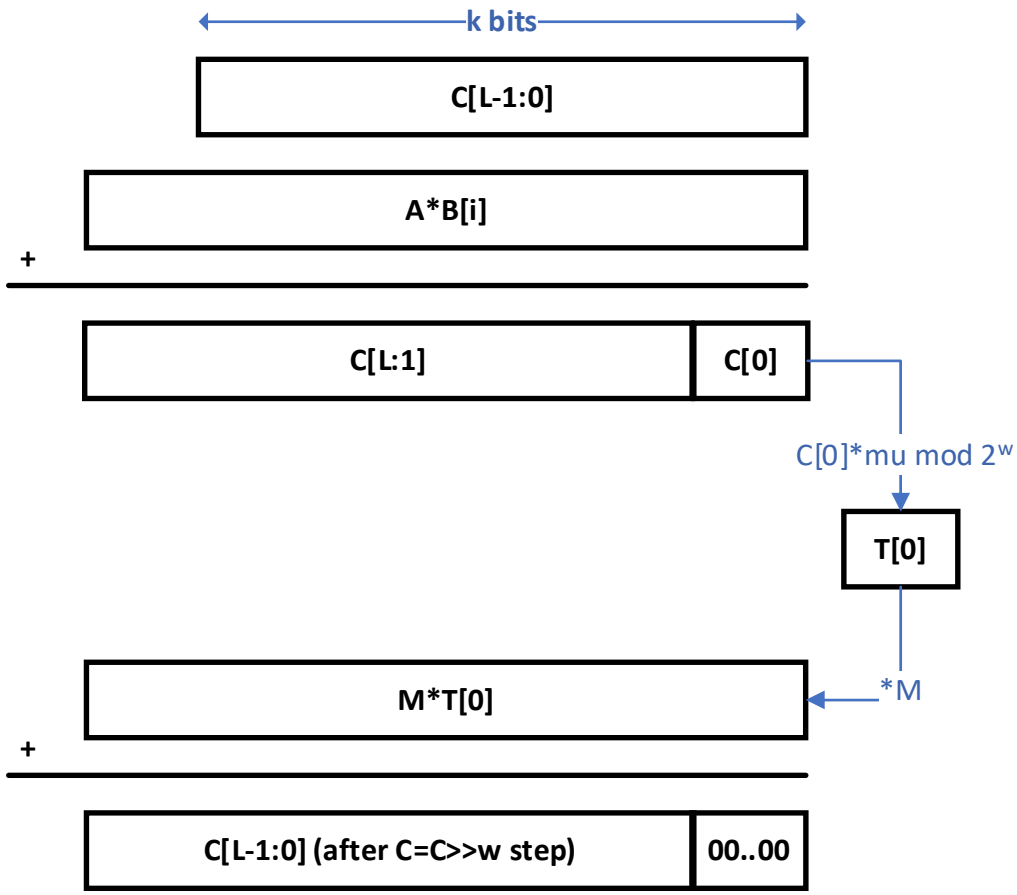


Figure 3. Single Iteration of "Almost" Montgomery Multiplication Operation

## 2.3 Technology Description

### 2.3.1 Almost Half Montgomery Multiplication

The Almost Montgomery Multiplication algorithm is defined as follows (Assume L is even):

| | |
|---|---|
| **Inputs:** | A,B,M,mu |
| **Output:** | RES=AMM(A,B,M)=A*B*$R^{-1}$ mod M ($R=2^{L*w}$ mod M) |
| **Operation:** | AL=A[L/2-1:0]<br>AH=A[L-1,L/2]<br><br>C=AL*B+AH*B*$2^{(L/2)*w}$<br>for i from 0 to (L-1):<br>T=C[0]*mu // discard T[1]<br>C=C+T[0]*M // C[0] is zero<br>C=C>>w // at each step of the for loop, divide the result by $2^w$. |

For this algorithm, the output RES=AMM(A,B,M)=A*B*R^(-1) mod M can be rewritten as:

$$RES=A*B*R^{-1} \bmod M=(AL*B+AH*B*2^{(L/2)*w})*(2^{-L*w}) \bmod M$$
$$=(AL*B*(2^{-L*w}) \bmod M)+(AH*B*(2^{(L/2)*w})*(2^{-L*w}) \bmod M) \bmod M$$
$$=(AL*B*(2^{-L*w}) \bmod M)+(AH*B*(2^{-(L/2)*w}) \bmod M) \bmod M$$

Now, we utilize a precomputed constant K:

$$K=B*(2^{-(L/2)*w}) \bmod M$$

and rewrite the equation as follows:

$$RES \equiv A*B*R^{-1} \bmod M = (AL*B*(2^{-L*w}) \bmod M) + (AH*B*(2^{-(L/2)*w}) \bmod M) \bmod M$$
$$= (AL*B*(2^{-(L/2)*w})*(2^{-(L/2)*w}) \bmod M) + (AH*B*(2^{-(L/2)*w}) \bmod M) \bmod M$$
$$= (AL*K*(2^{-(L/2)*w}) \bmod M) + (AH*B*(2^{-(L/2)*w}) \bmod M) \bmod M$$
$$= (AL*K+AH*B)*(2^{-(L/2)*w}) \bmod M$$

With this precomputed K value, the "Almost" Half Montgomery Multiplication algorithm can be defined as follows:

| | |
|---|---|
| **Inputs:** | A,B<br>M,mu,K(K=B*($2^{-(L/2)*w}$) mod M) |
| **Output:** | **AHMM(A,B,M)**=A*B*$R^{-1}$ mod M |
| **Operation:** | AL=A[L/2-1:0]<br>AH=A[L-1,L/2]<br><br>C=AL*K+AH*B<br>for i from 0 to (L/2-1):<br>T=C[0]*mu // discard T[1]<br>C=C+T[0]*M // C[0] is zero<br>C=C>>w // at each step of the for loop, divide the result by $2^w$. |

This algorithm is depicted in Figure 4.

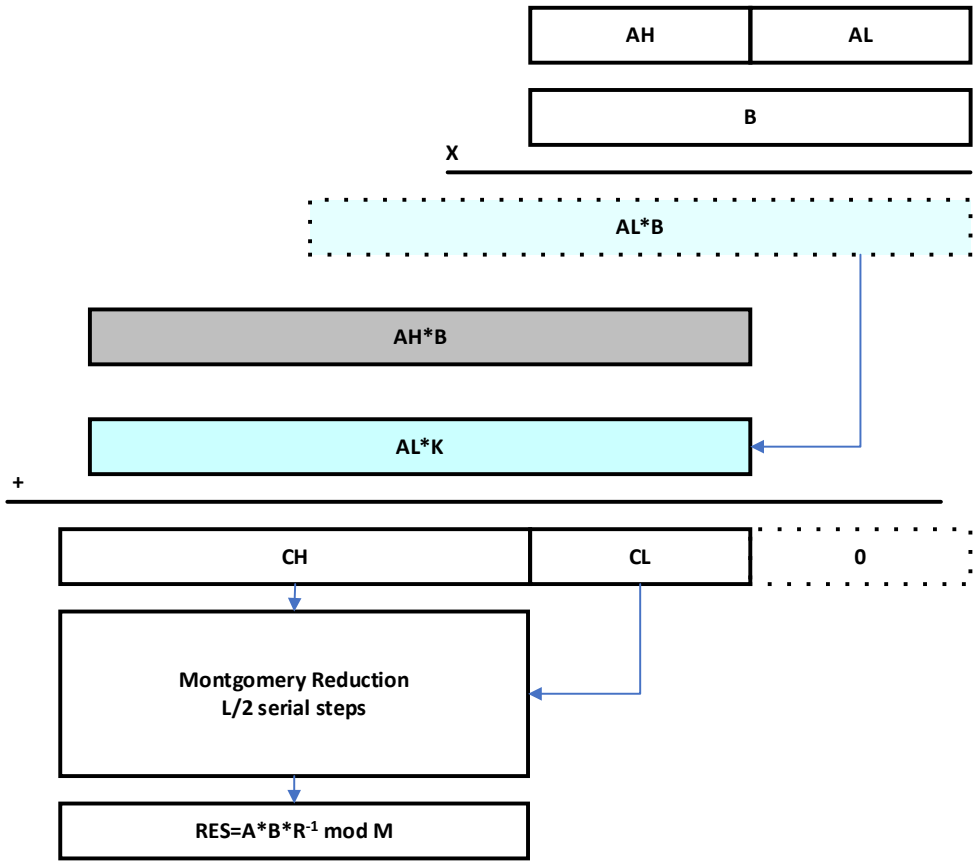


Figure 4. Almost Half Montgomery Multiplication

Our final word-level Almost Half Montgomery Multiplication Algorithm is defined as follows, which interleaves multiplication and reduction components:

| | |
|---|---|
| **Inputs:** | A,B,M,mu,K(K=B*($2^{-(L/2)*w}$) mod M) |
| **Output:** | **AHMM(A,B,M)**=A*B*$R^{-1}$ mod M |
| **Operation:** | AL=A[L/2-1:0]<br>AH=A[L-1,L/2]<br><br>C=0<br>for i from 0 to (L/2-1):<br>C=C+AL[i]*K+AH[i]*B<br>T=C[0]*mu // discard T[1]<br>C=C+T[0]*M // C[0] is zero<br>C=C>>w // at each step of the for loop, divide the result by $2^w$. |

# 3 Performance Result

## 3.1 Platform Configuration

Follow the software configuration steps in the below mentioned order:

1. OpenSSL
2. Intel® Integrated Performance Primitives Cryptography (Intel® IPP Cryptography)
3. Intel Multi-Buffer Crypto for IPsec
4. Intel QAT Engine

All components are installed in the local user directory and the system wide configuration is not affected.

***Note:*** All commands should be run within the start at "/home/rsa-user" directory.

### 3.1.1 OpenSSL

The Intel® IPP Cryptography component depends on OpenSSL version >= 3.0.8. If the OpenSSL version available on the platform is older than 3.0.8 then follow steps in the subsequent subsections to download and install the required OpenSSL version.

#### 3.1.1.1 Download and Install

```
> wget https://github.com/openssl/openssl/releases/download/openssl-3.1.3/openssl-3.1.3.tar.gz
> tar xzf openssl-3.1.3.tar.gz
> cd openssl-3.1.3
> ./Configure
> make -j
> export MYSSL=/home/rsa-user/openssl-3.1.3
> mkdir /home/rsa-user/myssl3
> make install DESTDIR=/home/rsa-user/myssl3
> export MYSSL3="/home/rsa-user/myssl3/usr/local"
> cd $MYSSL3/usr/local
> ln -s lib64 lib
```

#### 3.1.1.2 Check

```
env LD_LIBRARY_PATH=$MYSSL3/lib64 $MYSSL3/bin/openssl version -a
OpenSSL 3.1.3 19 Sep 2023 (Library: OpenSSL 3.1.3 19 Sep 2023)
```

### 3.1.2 Intel® Integrated Performance Primitives Cryptography (Intel® IPP Cryptography)

#### 3.1.2.1 Clone

```
git clone --recursive https://github.com/intel/ipp-crypto
mkdir /home/rsa-user/myipp
export MYIPP=/home/rsa-user/myipp
```

#### 3.1.2.2 Checkout and Compile Baseline Version

```
> git checkout 47079e5d3cd04a861c3d6a6985c6fc9985f90b7f
> CC=gcc CXX=g++ cmake CMakeLists.txt -B_build_baseline -DARCH=intel64 -
DOPENSSL_INCLUDE_DIR="${MYSSL3}/include" -DOPENSSL_LIBRARIES="${MYSSL3}/lib64" -
DOPENSSL_ROOT_DIR=”${MYSSL3}”
> cd _build_baseline
> make -j
> cmake --install . --prefix ${MYIPP}
```

#### 3.1.2.3 Checkout and Compile New Version

```
> git checkout 36e76e2388f3dd10cc440e213dfcf6ef59a0dfb8
> CC=gcc CXX=g++ cmake CMakeLists.txt -B_build_new -DARCH=intel64 -
DOPENSSL_INCLUDE_DIR="${MYSSL3}/include" -DOPENSSL_LIBRARIES="${MYSSL3}/lib64" -
DOPENSSL_ROOT_DIR=”${MYSSL3}”
> cd _build_new
> make -j
> cmake --install . --prefix ${MYIPP}
```

### 3.1.3 Intel Multi-Buffer Crypto for IPsec (intel-ipsec-mb)

#### 3.1.3.1 Download and Install

```
> wget https://github.com/intel/intel-ipsec-mb/archive/refs/tags/v1.4.tar.gz
> tar xzf v1.4.tar.gz
> cd intel-ipsec-mb-1.4
> make -j
> make install
```

### 3.1.4 Intel QAT Engine

#### 3.1.4.1 Download, Configure and Compile

```
> wget https://github.com/intel/QAT_Engine/archive/refs/tags/v1.3.0.tar.gz
> tar xzf v1.3.0.tar.gz
> cd QAT_Engine-1.3.0/
> export OPENSSL_ENGINES="${MYSSL3}/lib64/engines-3"
> export OPENSSL_MODULES="$(MYSSL3}/lib64/ossl-modules"
> ./configure --with-openssl_install_dir="${MYSSL3}" --with-openssl_dir="${MYSSL}" --enable-
qat_sw --disable-qat_hw --with-qat_sw_crypto_mb_install_dir="${MYIPP}" --disab
le-qat_sw_ecx --disable-qat_sw_ecdsa --disable-qat_sw_ecdh --disable-qat_sw_sm2 --disable-
qat_sw_gcm --disable-qat_sw_sha2 --disable-qat_sw_sm3 --disable-qat_sw_sm4_gcm --disable-
qat_sw_sm4_cbc --disable-qat_sw_sm4_ccm --enable-qat_provider --with-ld_opt="-
L${MYIPP}/lib/intel64/"
> make -j
> make install
> export LD_LIBRARY_PATH="${MYSSL3}/lib64:${MYSSL3}/lib64/ossl-modules:${MYIPP}/lib/intel64"
```

Verify key environment variables before progressing with performance testing.

```
> export
declare -x LD_LIBRARY_PATH="/home/rsa-user/myssl3/usr/local/lib64:/home/rsa-
user/myssl3/usr/local/lib64/ossl-modules:/home/rsa-user/myipp/lib/intel64"
declare -x MYIPP="/home/rsa-user/myipp"
declare -x MYSSL="/home/rsa-user/openssl-3.1.3"
declare -x MYSSL3="/home/rsa-user/myssl3/usr/local"
declare -x OPENSSL_ENGINES="/home/rsa-user/myssl3/usr/local/lib64/engines-3"
declare -x OPENSSL_MODULES="/home/rsa-user/myssl3/usr/local/lib64/ossl-modules"
```

## 3.2 OpenSSL Speed Data

### 3.2.1 Baseline RSA2048 Version[1]

```
> taskset -c 2,4 $MYSSL3/bin/openssl speed -provider qatprovider -elapsed -async_jobs 8 rsa2048
QAT_SW RSA for Provider Enabled
You have chosen to measure elapsed time instead of user CPU time.
Doing 2048 bits private rsa's for 10s: 78096 2048 bits private RSA's in 10.00s
Doing 2048 bits public rsa's for 10s: 1257170 2048 bits public RSA's in 10.01s
```

[1] See backup and appendix for workloads and configurations. Results may vary.

```
version: 3.1.3
built on: Mon Oct  2 10:08:42 2023 UTC
options: bn(64,64)
compiler: gcc -fPIC -pthread -m64 -Wa,--noexecstack -Wall -O3 -DOPENSSL_USE_NODELETE -DL_ENDIAN
-DOPENSSL_PIC -DOPENSSL_BUILDING_OPENSSL -DNDEBUG
CPUINFO: OPENSSL_ia32cap=0x7ffef3ffffebffff:0xfb417ffef3bfb7ef
                  sign    verify    sign/s verify/s
rsa 2048 bits 0.000128s 0.000008s   7809.6 125591.4
```

### 3.2.2 New RSA2048 Version[2]

```
> taskset -c 2,4 $MYSSL3/bin/openssl speed -provider qatprovider -elapsed -async_jobs 8 rsa2048
QAT_SW RSA for Provider Enabled
You have chosen to measure elapsed time instead of user CPU time.
Doing 2048 bits private rsa's for 10s: 85080 2048 bits private RSA's in 10.00s
Doing 2048 bits public rsa's for 10s: 1267008 2048 bits public RSA's in 10.00s
version: 3.1.3
built on: Mon Oct  2 10:08:42 2023 UTC
options: bn(64,64)
compiler: gcc -fPIC -pthread -m64 -Wa,--noexecstack -Wall -O3 -DOPENSSL_USE_NODELETE -DL_ENDIAN
-DOPENSSL_PIC -DOPENSSL_BUILDING_OPENSSL -DNDEBUG
CPUINFO: OPENSSL_ia32cap=0x7ffef3ffffebffff:0xfb417ffef3bfb7ef
                  sign    verify    sign/s verify/s
rsa 2048 bits 0.000118s 0.000008s   8508.0 126700.8
```

### 3.2.3 OpenSSL Speed RSA2048 Summary

As summarized in Figure 5, the new RSA2048 implementation offers about 8.9% better performance for signs per second vs the baseline RSA2048 version as measured by the OpenSSL speed tool (see 3.2.1 and 3.2.2).

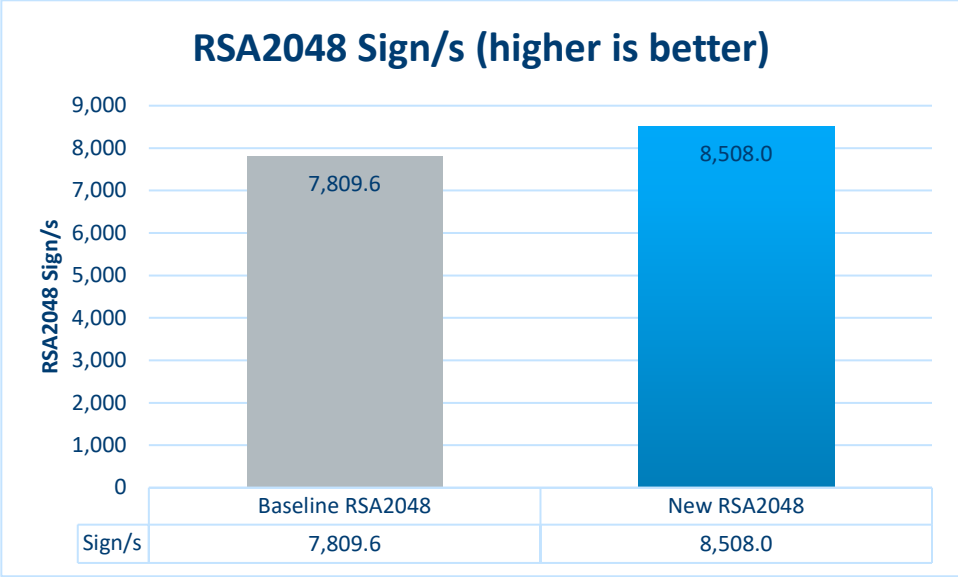


Figure 5. RSA2048 Sign/s Performance Results[3] (higher is better).

## 4 Benefits

This paper introduces a novel modular multiplication technique for large integer arithmetic. Our new algorithm allows efficient implementations of modular exponentiation, which improves overall RSA performance. The proposed software optimization for RSA2048 achieves up to a 1.089 gain factor vs OpenSSL on 4th Generation Intel® Xeon® Scalable processors. This optimization contributes to improvement of efficiency and responsiveness of Transport Layer Security (TLS) or Web servers as well as Internet Key Exchange (IKE) servers, since RSA2048 is the dominantly used algorithm. The "Top 1 Million Analysis - November 2021" report clearly shows that RSA and RSA2048 in particular is the predominant public key. With time, the popularity of RSA2048 is expected to drop to make way for more secure public keys. This trend is also visible in the 2021 report.

[2] See backup and appendix for workloads and configurations. Results may vary.

# 5 Summary

This paper studies efficient software implementations of modular multiplication. A novel modular multiplication technique is discussed and applied to Montgomery Multiplication/Reduction algorithms to optimize overall RSA performance. As proof of concept, an RSA2048 implementation is presented. Algorithmic and software characteristics of the optimizations are presented and discussed. Overall, the new technique achieves 1.089X faster RSA2048 sign operation over OpenSSL implementation.

The proposed RSA2048 optimization has been implemented in Intel® IPP Cryptography and can be used in production using Intel® QuickAssist Technology Engine for OpenSSL* (Intel® QAT Engine for OpenSSL*). The optimization idea can also be extended onto other RSA variants, i.e. RSA3072, RSA4096 or RSA8192.

# Appendix A System Configuration

Table 5. Hardware Configuration

| | |
|---|---|
| **System Summary** | 1-node, 2x Intel(R) Xeon(R) Gold 6454S, 32 cores, HT On, Turbo Off, NUMA 2, Integrated Accelerators Available [used]: DLB 8 [0], DSA 8 [0], IAA 0 [0], QAT 8 [0], Total Memory 32GB (1x32GB DDR5 4800 MT/s [4800 MT/s]); 16GB (1x16GB DDR5 4800 MT/s [4800 MT/s]), BIOS EGSDCRB1.86B.0081.D18.2205301332, microcode 0xaa000060, 1x Ethernet Controller I225-LM, 2x Ethernet Controller E810-C for QSFP, 1x 465.8G ST500DM002-1BD14, Ubuntu 22.04.3 LTS, 5.15.0-84-generic<br>Test by Intel as of 10/09/23. |

Table 6. Software Configuration

| | |
|---|---|
| GCC version | 11.4.0 |
| NASM version | 2.15.05 |
| OpenSSL version<br>https://github.com/openssl/openssl | 3.1.3 |
| Intel IPP Crypto (baseline)<br>https://github.com/intel/ipp-crypto | Commit id 47079e5d3cd04a861c3d6a6985c6fc9985f90b7f |
| Intel IPP Crypto (new)<br>https://github.com/intel/ipp-crypto | Commit id 36e76e2388f3dd10cc440e213dfcf6ef59a0dfb8 |
| Intel Multi-Buffer Crypto for IPsec<br>https://github.com/intel/intel-ipsec-mb | 1.4.0 |
| Intel QAT Engine<br>https://github.com/intel/QAT_Engine | 1.3.0 |



0324/DN/WIT/PDF 812656-001US