IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 5, 2019, accepted November 25, 2019, date of publication January 3, 2020, date of current version January 16, 2020.

Digital Object Identifier 10.1109/ACCESS.2019.2963826

Implementation of RSA Signatures on GPU and CPU Architectures

EDUARDO OCHOA-JIMÉNEZ[1], LUIS RIVERA-ZAMARRIPA[2], NARELI CRUZ-CORTÉS[2], (Member, IEEE), AND FRANCISCO RODRÍGUEZ-HENRÍQUEZ[1], (Member, IEEE)

[1]Computer Science Department, Cinvestav, Mexico City 07360, Mexico
[2]Centro de Investigación en Computación, Instituto Politécnico Nacional, Mexico City 07738, Mexico

Corresponding author: Luis Rivera-Zamarripa (lriveraz@gmail.com)

This work was supported in part by Instituto Politecnico Nacional de Mexico.

ABSTRACT This paper reports a constant-time CPU and GPU software implementation of the RSA exponentiation by using algorithms that offer a first-line defense against timing and cache attacks. In the case of GPU platforms the modular arithmetic layer was implemented using the Residue Number System (RNS) representation. We also present a CPU implementation of an RNS-based arithmetic that takes advantage of the parallelism provided by the Advanced Vector Extensions 2 (AVX2) instructions. Moreover, we carefully analyze the performance of two popular RNS modular reduction algorithms when implemented on many- and multi-core platforms. In the case of CPU platforms we also report that a combination of the schoolbook and Karatsuba algorithms for integer multiplication along with Montgomery reduction, yields our fastest modular multiplication procedure. In comparison with previous literature, our software library achieves faster timings for the computation of the RSA exponentiation using 1024-, 2048- and 3072-bit private keys.

INDEX TERMS Public key cryptography, RSA, RNS arithmetic, GPU, CPU, AVX2 instructions.

I. INTRODUCTION

Public key cryptosystems play an important role in communication systems that require the exchange of sensitive information. Proposed by Rivest, Shamir and Adleman in 1978 [1], RSA has become the most deployed public key cryptosystem in practical applications. The signing/verification of digital certificates is a heavily used application of RSA, as an important fraction of commercial digital certificates have been created using RSA as their cryptographic engine. However, due to its relatively high latency, RSA must be carefully implemented to achieve reasonable timing performance, memory and code footprints. Moreover, the computation of RSA main primitives, quite especially modular exponentiation, must be run in constant-time. This feature presents a first line of defense against timing and cache attacks [2].

The RSA instantiation using 1024-bit keys (a.k.a RSA-1024), has been widely used for computers on networks and traffic handling across the Internet. For applications requiring to achieve the 112- and 128-bit security levels, the National Institute of Standards and Technology (NIST) recommended in [3] the usage of RSA-2048 and RSA-3072, respectively. This recommendation should be contrasted with the newest analysis of state-of-the-art integer factorization algorithms [4], [5], which estimate that RSA-1024 and RSA-2048 can barely achieve the 76- and 106-bit security levels, respectively.

The RSA algorithm produces a public/private pair of keys by first constructing a per-user unique $2k$-bit modulo $N = p \cdot q$, where p, q are two k-bit prime numbers. The RSA public key is the tuple composed by the modulus N and a public exponent e, which is generally chosen as $e = 2^{16} + 1$. The RSA private exponent is defined as $d = e^{-1} \bmod \phi(N)$, where $\phi(\cdot)$ stands for the Euler's totient function. Given the RSA private key (d, N) and a message m, the Full Domain Hash (FDH) signature s of m is computed as $s = H(m)^d \bmod N$, where $H(\cdot)$ represents a hash function that maps m to $\mathbb{Z}_N$. It has been shown that the FDH RSA signature is provably secure [6]. A standard trick based on the Chinese Remainder Theorem trades the $2k$-bit RSA exponentiation $s = H(m)^d \bmod N$, by the computation of two independent k-bit modular exponentiations of the form, $s_1 = h^{d \bmod (p-1)} \bmod p$ and $s_2 = h^{d \bmod (q-1)} \bmod q$, where s_1, s_2 can be calculated concurrently.

The associate editor coordinating the review of this manuscript and approving it for publication was Chien-Ming Chen.

In this work, we focus our attention on the efficient parallel computation of s_1 and s_2 in GPU and CPU software implementations.

Most Internet transactions are executed using desktop computers, laptops and smartphones that are powered by multi-core micro-architectures based on general purpose Central Processing Units (CPUs). On the other hand, taking advantage of their massive parallelism, General Processing Units (GPUs) platforms have become an interesting option to speedup high demanding computational tasks such as the computation of several public key cryptographic primitives.

OUR CONTRIBUTIONS: In this work, two RSA constant-time software implementations for 1024-, 2048-, and 3072-bit RSA keys, are presented.

Our CPU software implementation of RSA uses a combination of integer arithmetic algorithms and Montgomery reduction that helped us to exploit the fine-grained parallelism present in the latest Intel micro-architectures. We also took advantage of the multi-core architecture of modern Intel CPU processor to concurrently compute two RSA exponentiations. Further, we explore the usage of the Advanced Vector Extension 2 (AVX2) for achieving an efficient Residue Number System (RNS) field arithmetic, as an alternative approach for the parallel computation of the RSA signature. Likewise, our RSA GPU implementation also employs RNS arithmetic, which permits to take a better advantage of the massive parallelism available on this many-core architecture.

Our software implementation targeted four platforms, namely, a GPU GeForce GTX TITAN running on a Kepler architecture at 876 MHz, a GPU GeForce GTX 1080 running on a Kepler architecture at 1.81 GHz and a CPU Intel core i7 equipped with Haswell and Skylake micro-architectures running at 2.6 GHz, 4 GHz and 1.8GHz, respectively.

Our experimental results show that computing one RSA signature takes far less time when calculated using our CPU software library. However, our GPU software scales better for larger RSA key lengths, and offers better performance when not one but many RSA signatures must be computed at once.

The experimental results presented in this work outperform previously reported GPU RSA implementations [7]–[10] by a factor of 1.24, 1.27 and 2.98 for RSA-1024 bits, RSA-2048 bits, and RSA-3072, respectively. Regarding our CPU implementation of RSA, our results outperform previous CPU software implementations [11], [12] by a factor of 1.84 and 1.15 and 1.19 for the RSA-1024, RSA-2048 and RSA-3072, respectively.

The remainder of this paper is organized as follows. In §II a review of the main modular arithmetic algorithms used in this work is given. Then, in §III and §IV we present a detailed description of the CPU and GPU implementations of the RSA exponentiation, respectively. Finally, we draw some concluding remarks in §V.

II. ARITHMETIC BACKGROUND

One of the main objectives of this work is to perform a fast and constant-time modular exponentiation, which is required by the RSA signature algorithm. Hence, we start this section by describing in §II-A the regular-recoding exponentiation algorithm used for performing the RSA private operation. Furthermore, throughout this work we use two different approaches for computing the underlying modular arithmetic. The first approach adopts the Montgomery representation discussed in §II-B, whereas the second one uses an arithmetic layer based on the Residue Number System (RNS) representation as explained in §II-C. In this section, we present an overview of these two arithmetic representations.

NOTATION: Let N be a $2k$-bit RSA modulus of the form $N = p \cdot q$, where p, q are two k-bit prime numbers. Since an RSA exponentiation modulo N can be traded by two k-bit exponentiations modulo p and q, in this work we focus our attention on the computation of the operation $y = x^e \bmod p$, where it will be assumed that all the operands have a bit-length of k bits. The integer representation of a k-bit integer can be accommodated in $n = \lceil \frac{k}{w} \rceil$ words, where each word has a size of w bits. Throughout this work, word sizes of $w = 32$ and $w = 64$ bits will be assumed. An element $a \in \mathbb{Z}_p$ is represented in radix-r as the array $a = \sum_{i=0}^{n-1} a_i r^i$, where $r = 2^w$ and $0 \leq a_i < r$. We say that the operand a has a word-length of n words. For the sake of simplicity, we will only consider operands with an even word-length. Particularly, we are interested in the cases $n = 8, 16, 24$, required for computing RSA-1024, RSA-2048 and RSA-3072 signatures, respectively.

A. CONSTANT-TIME MODULAR EXPONENTIATION

We adopted a variant of the fixed-window exponentiation method, which starts by producing a regular recoding of the exponent. To this aim, we use the procedure proposed by Joye and Tunstall in [13] as shown in Algorithm 1. Given a k-bit exponent, Algorithm 1 provides an encoding of length $\eta = \lceil \frac{k}{\omega} \rceil + 1$, whose digits belong to the set $\{1, 2, \ldots, 2^\omega\}$, where ω is the prescribed window size.

Algorithm 1 Unsigned Exponent Regular Recoding [13]

Require: A k-bit exponent e, window size ω.
Ensure: $f = (f_{\eta-1}, \ldots, f_0)$ with $f_i \in \{1, 2, \ldots, 2^\omega\}$ for $0 \leq i < \eta$.

1: $i \leftarrow 0 \quad j \leftarrow 1$
2: **while** $e \geq 2^\omega + 1$ **do**
3: $\quad d \leftarrow e \mod 2^\omega$
4: $\quad d' \leftarrow d + j + 2^\omega - 2$
5: $\quad f_i \leftarrow (d' \mod 2^\omega) + 1$
6: $\quad j \leftarrow \lfloor d'/2^\omega \rfloor$
7: $\quad e \leftarrow \lfloor e/2^\omega \rfloor$
8: $\quad i \leftarrow i + 1$
9: **end while**
10: $f_i \leftarrow e + j - 1$
11: **return** f

Algorithm 2 allows us to perform a constant-time modular exponentiation, which helps us to thwart basic timing side-channel attacks. Since this windowed exponentiation

algorithm requires to lookup a pre-computed table, it is necessary to implement a mechanism that protects the corresponding queries. Hence, whenever the pre-computed table Γ is queried in Step 6, a *linear pass* memory access is performed as a protective countermeasure [14]. This technique consists of traversing the entire pre-computed table Γ every time that a certain position is accessed. In this way we ensure that all memory queries have the same running time. The protected modular exponentiation that computes $y = x^e \bmod p$ is shown in Algorithm 2. This algorithm has a cost of exactly $\lfloor \frac{k}{\omega} \rfloor$ modular multiplications and $k-1$ modular squaring operations.

Algorithm 2 Protected Fixed-Window Modular Exponentiation

Require: The k-bit integers x, e and p, and the window size ω.
Ensure: A k-bit integer y such that $y = x^e \bmod p$.
Precomputation:
1: Recode e using Algorithm 1 to obtain the encoding f of length $\eta = \lceil \frac{k}{\omega} \rceil + 1$.
2: Compute $\Gamma[i] \leftarrow x^i \bmod p$ for $i \in \{0, \ldots, 2^\omega\}$
Computation:
3: $y \leftarrow$ Perform a *linear pass* to recover $\Gamma[f_{\eta-1}]$
4: **for** $i = \eta - 2$ **down to** 0 **do**
5: $\quad y \leftarrow y^{2^\omega}$
6: $\quad z \leftarrow$ Perform a *linear pass* to recover $\Gamma[f_i]$
7: $\quad y \leftarrow y \cdot z$
8: **end for**
9: **return** y

B. MONTGOMERY MODULAR ARITHMETIC

In 1985, Montgomery proposed a novel method to compute field multiplications without using trial divisions [15]. Montgomery suggested to change the operand representation to the so-called *Montgomery domain*.[1] Let us define the Montgomery parameter R as, $R = r^n$, where as before n represents the number of words necessary to represent the prime modulus p in radix $r = 2^w$. Hence, $r^{n-1} < p < r^n$. The Montgomery representation $\tilde{a}$ of an element $a \in \mathbb{Z}_p$ is computed as $\tilde{a} = a \cdot R \bmod p$.

Let as assume that the elements $a, b \in \mathbb{Z}_p$ have a Montgomery's representation given as $\tilde{a}$ and $\tilde{b}$, respectively. Let d be given as $d = \tilde{a} \cdot \tilde{b}$. Then, the Montgomery product of $\tilde{a}$ and $\tilde{b}$ is defined as $\tilde{c} = \tilde{a} \cdot \tilde{b} \cdot R^{-1} \bmod p$, which can be readily computed as

$$\tilde{c} = \frac{d + (\mu \cdot d \mod R) \cdot p}{R} \equiv d \cdot R^{-1} \mod p, \quad (1)$$

where the parameter μ given as $\mu = -p^{-1} \bmod R$, can be pre-computed off-line. Also the reduction and division by R operations, can be efficiently performed using fast right/left n-word shift operations. It can be shown that when $0 \leq d < p^2$, the result $\tilde{c}$ of Equation (1) is an integer in the interval $[0, 2p[$. Hence, at most a single conditional subtraction is needed to obtain $0 \leq \tilde{c} < p$. This conditional subtraction must be performed in a constant-time fashion.

[1] Usually at the start and at the end of the RSA computation, operand transfers to and from the Montgomery representation are performed.

C. RNS MODULAR ARITHMETIC

In the nineties of the last century, several authors proposed the usage of the Residue Number System (RNS) as an alternative for computing modular arithmetic over large integer operands [16]–[19].[2]

Taking advantage of the ancient Chinese Remainder Theorem (CRT), RNS main attractiveness is to represent a large integer by means of a set of smaller independent numbers. In this way, one trades the computational cost of a single arithmetic operation over two large operands by the calculation of independent smaller modular operations that may be computed in parallel. The RNS representation is defined as follows.

Let $\mathcal{B}$ be an RNS-basis consisting of a set of ℓ pairwise co-prime integer moduli $\mathcal{B} = \{m_1, m_2, \ldots, m_\ell\}$, and let $M = \prod_{i=1}^{\ell} m_i$. Then, an integer $a \in [0, M-1]$ can be uniquely represented by the ℓ-tuple $a = (a_1, a_2, \ldots, a_\ell)$, where each a_i is the residue of a modulo m_i. In the remainder of this paper this reduction operation will be written as $a_i = |a|_{m_i}$. From its RNS representation, the corresponding binary representation of a can be obtained using the following recovery formula,

$$a = \left| \sum_{i=1}^{\ell} \left| a_i \cdot M_i^{-1} \right|_{m_i} \cdot M_i \right|_M, \quad \text{where } M_i \triangleq M/m_i. \quad (2)$$

Let a and b be two large k-bit integers with $a, b < M$, represented as RNS tuples $a = (a_1, a_2, \ldots, a_\ell)$ and $b = (b_1, b_2, \ldots, b_\ell)$. Then, RNS addition $\oplus$ and RNS multiplication $\otimes$ are performed coefficient-wise as,

$$\begin{aligned} c = a \oplus b &= (c_1 = |a_1 + b_1|_{m_1}, c_2 = |a_2 + b_2|_{m_2}, \\ &\quad \ldots, c_\ell = |a_\ell + b_\ell|_{m_\ell}), \\ d = a \otimes b &= (d_1 = |a_1 \cdot b_1|_{m_1}, d_2 = |a_2 \cdot b_2|_{m_2}, \\ &\quad \ldots, d_\ell = |a_\ell \cdot b_\ell|_{m_\ell}). \end{aligned} \quad (3)$$

If the target platform is equipped with ℓ processing units, then the computational cost associated to any of the two RNS arithmetic operations of Eq. (3) is approximately the same as performing one single coefficient multiplication.

Remark 1: Let a, b be two k-bit integers. Then, the integer product $d = a \otimes b$ of Eq. (3) can be uniquely recovered from its RNS representation if and only if $d < M$. Since in general the integer product d is a $2k$-bit number, it follows that an RNS representation of a, b and d requires an RNS-basis composed of ℓ w-bit moduli, with $\ell \geq 2\lceil \frac{k}{w} \rceil = 2n$.

Remark 2: For the sake of efficiency, the integer moduli m_i are usually selected as $m_i = 2^w - \mu_i$, where μ_i coefficients are chosen as small as possible. If $\mu_i < 2^{\lfloor \frac{w}{2} \rfloor}$,

[2] See [20] for a recent survey on several RNS reduction strategies.

then the coefficients d_i for $i = 1, \ldots, \ell$ of Eq. (3) can be efficiently computed by repeating at most twice the operation, $t_i = d_i \bmod 2^w + \mu_i \cdot \lfloor d_i/2^w \rfloor$. Thereafter it is guaranteed that $t_i \in [0, 2^w[$. Since $2^w > m_i$, one may need to compute a final reduction, $d_i = |a_i \cdot b_i|_{m_i} = t_i \bmod m_i$, which can be achieved at a cost of at most one subtraction operation. In order to assure a constant-time implementation, this reduction is carried out by executing *exactly* two conditional reductions of the form, $t_i = a \bmod 2^w + \mu_i \cdot \lfloor a/2^w \rfloor$, followed by *exactly* one conditionally subtraction by m_i.

1) RNS MODULAR REDUCTION

A modular multiplication is often performed by first computing the integer multiplication $d = a \otimes b$, followed by a reduction modulo p so that the resulting value lies in the range $[1, p-1]$. In this work, we adopted the reduction approach proposed in [16], [18], and its adaptation to GPU platforms presented by Jeljeli in [21] (see also [22]). Crucially, this approach allows to perform a modular reduction $d \bmod p$ without leaving the RNS domain as it is explained next.

Let d be a large integer represented in RNS using a basis composed by ℓ one-word moduli. Let us assume that d must be reduced modulo an n-word prime number p. Then, a strategy to perform the modular reduction $d \bmod p$ can be obtained from a direct application of the RNS recovery formula of Eq. (2) as

$$d = \left| \sum_{i=1}^{\ell} \gamma_i \cdot M_i \right|_M = \left(\sum_{i=1}^{\ell} \gamma_i \cdot M_i \right) \bmod M,$$
$$\text{where } \gamma_i \triangleq \left| d_i \cdot M_i^{-1} \right|_{m_i}. \tag{4}$$

Notice that the coefficients γ_i for $i = 1, \ldots, \ell$ as defined in Eq. (4), can be seen as a single RNS vector γ with ℓ coordinates. Notice that d can also be written as

$$d = \sum_{i=1}^{\ell} \gamma_i \cdot M_i - \alpha \cdot M, \tag{5}$$

where α is some positive integer, and by construction, $0 \leq d/M < 1$. From Eq. (5), the parameter α can be estimated as

$$\alpha \approx \left\lfloor \sum_{i=1}^{\ell} \frac{\gamma_i}{m_i} \right\rfloor.$$

Since $\gamma_i < m_i$, we have that $0 \leq \alpha < \ell$. Observe that the value

$$z = \sum_{i=1}^{\ell} \gamma_i \cdot |M_i|_p - |\alpha \cdot M|_p, \tag{6}$$

is congruent to $d \bmod p$, but in general $z \geq d$. In order to obtain a good approximation of α, which at the same time can be computed efficiently, one uses the fact that $m_i \approx 2^w$. Hence, the ratio γ_i/m_i can be approximated by only considering the σ most significant bits of the quotient $\gamma_i/2^w$ as

$$\hat{\alpha} \triangleq \left\lfloor \sum_{i=1}^{\ell} \frac{\left\lfloor \frac{\gamma_i}{2^{w-\sigma}} \right\rfloor}{2^{\sigma}} + \Delta \right\rfloor, \tag{7}$$

where σ is an integer in the range $[1, w]$ and $0 < \Delta < 1$ is an error correcting parameter.

Remark 3: The integer part of the summation in Eq. (7) can be efficiently computed by considering the output carry c produced by the addition of the σ most significant bits of the coefficients γ_i with $i = 0, 1, \ldots, \ell$. Notice that the output carry c is an integer in the range $[0, \ell[$.

Algorithm 3 computes the RNS vector $z \equiv d \bmod p$ as defined in Eq. (6). In Steps 4-6, ℓ processing units concurrently compute ℓ copies of the RNS vector γ given in Eq. (4). Although the computational cost of these steps is of ℓ RNS multiplications, their associated latency is very close to the latency associated to one RNS multiplication. The second loop of Algorithm 3 (Steps 7-13) completes the computation of the RNS vector z. Step 9 performs ℓ and $\ell - 1$ RNS multiplications and additions, respectively. As before, all these ℓ RNS multiplications can be computed in parallel, but they must be sequentially added using a binary tree adder. Step 10 computes α at the cost of adding ℓ σ-bit integers (cf. Remark 3). In Step 11, the RNS vector z is finally obtained by performing one RNS multiplication and one RNS subtraction.

Summarizing, the latency associated to Algorithm 3 is the combined latency of three RNS multiplications plus one RNS subtraction plus the addition of ℓ σ-bit numbers.

Algorithm 3 does not calculate $d \bmod p$, but instead produces an RNS multiple of it, which is bounded by $2^w \cdot \ell \cdot$ [21], [22]. In practice this implies that the RNS vector z must be accommodated using at least two extra safeguard moduli. Consequently, we increased the cardinality of the RNS basis $\mathcal{B}$ from ℓ to $\ell + 3$ moduli. By taking this caution measure, one guarantees that accumulating thousands of modular multiplications (required in the computation of a typical RSA exponentiation), will not exceed the RNS bound M.

2) RNS MONTGOMERY MODULAR REDUCTION

Alternatively, the modular reduction by a k-bit prime number p can be performed by adapting the Montgomery reduction given in Equation (1), to the RNS representation setting. This approach was first introduced by Posch and Posch [23], and several refinements were proposed in [24] and in a myriad of subsequent papers [20].

The adaptation of the k-bit Montgomery reduction to RNS arithmetic requires to handle two distinct RNS-basis $\mathcal{B} = \{m_1, m_2, \ldots m_\ell\}$ and $\mathcal{B}' = \{m'_1, m'_2, \ldots m'_\ell\}$ such that $\gcd(M, M') = \gcd(M, p) = 1$, where $\ell = \lceil \frac{k}{w} \rceil = n$, and $M = \prod_{i=1}^{\ell} m_i$ and $M' = \prod_{i=1}^{\ell} m'_i$. In addition, the Montgomery parameters of Equation (1) must be represented using two RNS bases $\mathcal{B}$ and $\mathcal{B}'$. It is customary to choose

Algorithm 3 RNS Modular Reduction Optimized for Multi/Many Core Platforms [21]

Require: The integer d given in ℓ-moduli RNS representation, the ℓ-moduli RNS-basis $\mathcal{B}$, and parameters r, σ, and Δ.

Ensure: RNS vector z, such that its integer representation is $z \equiv d \mod p$.

Precomputation:

1: RNS vector $\left|M_j^{-1}\right|_{m_j}$ for $j \in \{1, \ldots, \ell\}$
2: Table of RNS vectors $|M_i|_p$ for $i \in \{1, \ldots, \ell\}$
3: Table of RNS vectors $|\alpha \cdot M|_p$ for $\alpha \in \{1, \ldots, \ell - 1\}$
4: **for each** processing unit j **do**
5: $\quad \gamma_j \longleftarrow \left| d_j \cdot \left|M_j^{-1}\right|_{m_j} \right|_{m_j}$
6: **end for**
7: **for each** processing unit j **do**
8: $\quad$ **for each** processing unit i **do**
9: $\quad\quad z_j \longleftarrow \left| \sum_{i=1}^{\ell} \gamma_i \cdot \left||M_i|_p\right|_{m_j} \right|_{m_j}$ ▷ requires ℓ RNS mults and $\ell - 1$ RNS adds.
10: $\quad\quad \alpha \longleftarrow \left\lfloor \sum_{i=1}^{\ell} \frac{\left\lfloor \frac{\gamma_i}{2^{w-\sigma}} \right\rfloor}{2^{\sigma}} + \Delta \right\rfloor$
11: $\quad\quad z_j \longleftarrow \left| z_j - \left||\alpha \cdot M|_p\right|_{m_j} \right|_{m_j}$
12: $\quad$ **end for**
13: **end for**
14: **return** $z = (z_1, \ldots, z_\ell)$

$R = M$, instead of $R = r^n$. Moreover, the parameter μ of Equation (1) is represented by the RNS vector $-p^{-1} \bmod R$ represented in base $\mathcal{B}$.

As discussed in [24], [25], the RNS version of the Montgomery reduction can avoid a conditional subtraction by adopting Walter's approach of [26]. Indeed, a redundant representation of the elements in Montgomery representation can be achieved by choosing a Montgomery radix such that $4p < R$ and a RNS-basis $\mathcal{B}'$ such that $2p < M'$. At the end of the whole computation the result can be normalized at the cost of a single constant-time subtraction.

The procedure to compute an RNS Montgomery modular reduction is presented in Algorithm 4. It is noted that the multiplication $d_{\mathcal{B}}$ by μ in Step 5, is carried out in base $\mathcal{B}$. Due to the design choice $R = M$, the reduction modulo R is implicitly applied in this computation. Thus, the product performed in this step is equivalent to compute $\mu \cdot d \bmod R$ of Equation (1). The numerator of Equation (1) is computed in Step 11. This calculation must be performed in base $\mathcal{B}'$ because the expected result is a multiple of R, which is equal to zero in base $\mathcal{B}$.

A division by R is computed in Step 12. The output of this operation corresponds to the RNS representation of $d \cdot R^{-1} \bmod p$ in base $\mathcal{B}'$. This computation must be performed in base $\mathcal{B}'$ because R^{-1} is not defined in base $\mathcal{B}$. Thus, throughout the algorithm it becomes necessary to

Algorithm 4 RNS Montgomery Modular Reduction [24]

Require: The integer d given in the two bases ℓ-moduli RNS representations $d_{\mathcal{B}}$ and $d_{\mathcal{B}'}$, the ℓ-moduli RNS-basis $\mathcal{B}$ and $\mathcal{B}'$.

Ensure: The RNS vectors $z_{\mathcal{B}}$ and $z_{\mathcal{B}'}$ corresponding to the integer representation of $z \equiv d \mod p$.

Precomputation:

1: RNS vectors $\left|M_i^{-1}\right|_{m_i}$, $\left|M_i'^{-1}\right|_{m_i'}$, $\left|M^{-1}\right|_{m_i'}$, $\left|-p^{-1} \bmod M\right|_{m_i}$ and $|p|_{m_i'}$ for $i \in \{1, \ldots, \ell\}$
2: Table of vectors $|M_i|_{m_j'}$ and $\left|M_i'\right|_{m_j}$ for $i, j \in \{1, \ldots, \ell\}$
3: Tables of RNS vectors $|\alpha \cdot (-M)|_{m_i'}$ and $\left|\alpha \cdot (-M')\right|_{m_i}$ for $\alpha, i \in \{1, \ldots, \ell\}$

Computation:

4: **for each** processing unit i **do**
5: $\quad \gamma_i \longleftarrow \left| d_{\mathcal{B}i} \cdot \left|-p^{-1} \bmod M\right|_{m_i} \right|_{m_i}$ ▷ 1 RNS product
6: $\quad \theta_i \longleftarrow \left| \gamma_i \cdot \left|M_i^{-1}\right|_{m_i} \right|_{m_i}$ ▷ 1 RNS product
7: **end for**
8: $\alpha \longleftarrow \left\lfloor \sum_{j=1}^{\ell} \frac{\left\lfloor \frac{\theta_j}{2^{w-\sigma}} \right\rfloor}{2^{\sigma}} \right\rfloor$ ▷ Addition of ℓ σ-bit terms
9: **for each** processing unit i **do**
10: $\quad \delta_i \longleftarrow \left| \sum_{j=1}^{\ell} \left||M_i|_{m_j'} \cdot \theta_j\right|_{m_i'} + |\alpha(-M)|_{m_i'} \right|_{m_i'}$ ▷ ℓ RNS products and ℓ RNS additions
11: $\quad \gamma_i \longleftarrow \left| d_{\mathcal{B}'i} + \left(\delta_i \cdot |p|_{m_i'}\right) \right|_{m_i'}$ ▷ 1 RNS product and 1 RNS addition
12: $\quad z_{\mathcal{B}'i} \longleftarrow \left| \gamma_i \cdot \left|M^{-1}\right|_{m_i'} \right|_{m_i'}$ ▷ 1 RNS product
13: $\quad \theta_i \longleftarrow \left| z_{\mathcal{B}'i} \cdot \left|M_i'^{-1}\right|_{m_i'} \right|_{m_i'}$ ▷ 1 RNS product
14: **end for**
15: $\alpha \longleftarrow \left\lfloor \sum_{j=1}^{\ell} \frac{\left\lfloor \frac{\theta_j}{2^{w-\sigma}} \right\rfloor}{2^{\sigma}} + 0.5 \right\rfloor$ ▷ Addition of ℓ σ-bit terms
16: **for each** processing unit i **do**
17: $\quad z_{\mathcal{B}i} \longleftarrow \left| \sum_{j=1}^{\ell} \left|\left|M_i'\right|_{m_j} \cdot \theta_j\right|_{m_i} + \left|\alpha(-M')\right|_{m_i} \right|_{m_i}$ ▷ ℓ RNS products and ℓ RNS additions
18: **end for**
19: **return** $z_{\mathcal{B}}$ and $z_{\mathcal{B}'}$

perform two *base extensions*, which consist of transforming a number given in base $\mathcal{B}$ (resp. $\mathcal{B}'$) into a number in base $\mathcal{B}'$ (resp. $\mathcal{B}$.) The first base extension (Steps 6 to 10) is performed to derive an approximation for δ obtained from the value γ calculated in Step 5. This permits to compute the value $(d + (\mu \cdot d \bmod R) \cdot p)/R$ in base $\mathcal{B}'$. The second base extension (Steps 14 to 17) is performed at the end of the algorithm. This extension obtains the RNS representation of the final result in base $\mathcal{B}$ (which was computed in Step 12 in base $\mathcal{B}'$).

III. EFFICIENT IMPLEMENTATION OF RSA ON CPU PLATFORMS

In this section, efficient CPU software implementations of the RSA exponentiation are described in detail. First, the implementation of elementary arithmetic operations such as multiplication, squaring and Montgomery reduction, is described. Moreover, a brief explanation of how this modular arithmetic layer was used for the concurrent computation of the two exponentiations associated to the RSA signature is given. Thereafter, a description of the RNS-based arithmetic implementation is presented. The implementation of this arithmetic heavily relies on the set of Advanced Vector Extensions 2 `AVX2`. Furthermore, a comparison between the RNS reduction procedures described in Algorithm 3 and Algorithm 4 shows that for our CPU implementation setting, the latter is faster than the former.

All the experimental results presented in this section were executed on an Intel Core i7-4770 processor supporting the Haswell micro-architecture and on an Intel Core i7-6700 processor that supports the Skylake micro-architecture, equipped with 16 GB of RAM memory and using the Ubuntu 16.04.6 LTS operating system. To guarantee the reproducibility of our measurements, the Intel Hyper-Threading and Intel Turbo Boost technologies were disabled. Our source code was compiled using the GNU C Compiler (gcc) v6.1.0 with the `-O3` optimization flag and using the options `-mbmi2 -fwrapv -fomit-frame-pointer` and `-mbmi2 -madx -fwrapv -fomit-frame-pointer` for the Haswell and Skylake micro-architectures, respectively.[3]

A. MONTGOMERY BASED ARITHMETIC ON CPU PLATFORMS

1) INTEL INTEGER ARITHMETIC INSTRUCTIONS

Aiming to achieve efficient integer multiplication/squaring and Montgomery's reduction, we took advantage of the instruction `MULX` and the set of instructions Multi-precision Add-carry instruction extensions `ADX` [27]. First introduced in the Bit Manipulation instruction set (BMI2) of the Haswell micro-architecture, the assembly instruction `MULX` is an extension of the 64-bit multiplication instruction `MUL`. `MULX` uses a three-operand code and computes an unsigned multiplication without modifying the arithmetic flags. The advantage of the three-operand code is that permits to save `MOV` instructions by allowing to choose the output registers receiving the highest and the lowest part of the two-word output product. On the other hand, the instruction set `ADX` first introduced in the Broadwell micro-architecture, includes the instructions `ADCX` and `ADOX`, which were designed to handle two independent carry chains. These instructions compute unsigned 64-bit additions with input carry. The resulting output carry is stored in the carry flag (`CF`) and the overflow flag (`OF`), respectively. Since both instructions deal with two independent carry chains, they can be executed in parallel. An important advantage of these instructions is that they allow to combine `MULX`, `ADC`, `ADCX` and `ADOX` instructions without corrupting the carry chain. This feature has a noticeable impact in the efficiency of the Montgomery based arithmetic as discussed next.

2) INTEGER MULTIPLICATION

The two most popular approaches for computing integer multiplication in a software implementation are the schoolbook method with its associated quadratic complexity on the number of word multiplications, and the Karatsuba and Ofman [28] approach that enjoys a sub-quadratic complexity on the number of word multiplications at the price of increasing the number of required word additions. One can also opt for using a combination of these two methods, which was the strategy adopted in this work.

The efficiency of the schoolbook method mainly depends on how the partial products are computed and the way that they are added. We used the operand-scanning strategy, where the multiplicand operand is multiplied by each word of the multiplier. This strategy allows us to take full advantage of the `MULX`, `ADCX` and `ADOX` instructions.

However, this approach is limited by the available number of general purpose registers. Because of this, the schoolbook method tends to be efficient only when the operands have a small word-length. Indeed, as shown in Table 1, our implementation of a pure schoolbook integer multiplication outperformed Karatsuba only when the operands had a word-length in the range $0 < n \leq 8$.

Therefore, n-word multiplications with $n > 8$ were performed using a combination of the Karatsuba multiplication method [28] and the schoolbook approach. Two n-word integers a and b can be written as $a = a_0 + a_1 \cdot r^{n/2}$ and $b = b_0 + b_1 \cdot r^{n/2}$, where as before $r = 2^w$. Using the Karatsuba approach, one first computes the values $c_L = a_0 \cdot b_0$, $c_M = (a_0 + a_1)(b_0 + b_1)$ and $c_H = a_1 \cdot b_1$. Then the integer multiplication $c = a \cdot b$ is obtained as

$$c = c_L + (c_M - c_L - c_H) \cdot r^{n/2} + C_H \cdot r^n.$$

This computation costs two additions and three multiplications of $n/2$-word integers, plus one addition and two subtractions of n-word operands.

For 16-word multiplications, we applied one Karatsuba level that took us from 16- to 8-word multiplications. In the case of 24-word multiplications, we utilized two Karatsuba levels that took us from 24- to 12-word multiplications, and then to 6-word multiplications. The results obtained from this strategy are presented in Table 1.

COMPARISON AGAINST SCOTT'S KARATSUBA VARIANT

In [29], Scott proposed a Karatsuba variant based on Arbitrary degree Karatsuba (ADK) previously suggested in [30]. Scott implemented the ADK approach in the *reduced-radix* setting, where a number is represented using a word size lower than the one belonging to the target processor.

[3]The source code of our software library is available at, `https://github.com/luinxz/RSA`.

TABLE 1. A comparison of integer multiplication using Karatsuba and schoolbook method. Computational costs are reported in number of word multiplications (using `MULX` instructions) and clock cycles measured on Haswell (HW) and Skylake (SK) micro-architectures.

n	MULX		Clock cycles			
	Karatsuba	schoolbook	Karatsuba		schoolbook	
			HW	SK	HW	SK
2	3	4	20	14	12	8
3	9	9	28	28	20	16
4	12	16	60	43	32	24
6	27	36	112	87	84	48
8	48	64	196	137	184	87
12	121	-	400	278	-	-
16	209	-	692	419	-	-
24	376	-	1328	960	-	-

TABLE 2. Comparison of timings for integer multiplication using Scott strategy [29] against Karatsuba-schoolbook method. All timings are reported in clock cycles measured on Haswell (HW) and Skylake (SK) micro-architectures.

n-word multiplication	Clock cycles			
	Scott [29]		Schoolbook + Karatsuba	
	HW	SK	HW	SK
2×2	36	38	12	8
3×3	56	58	20	16
4×4	76	77	32	24
6×6	128	119	84	48
8×8	228	189	184	87
12×12	376	312	400	278
16×16	600	503	692	419
24×24	1224	1006	1328	960

The advantage of this strategy is that the partial products can be accumulated without worrying about the output carries.

We implemented Scott's strategy using a word size of $r = 2^{62}$ bits as proposed in [29]. A comparison of our own implementation of Scott's proposal against the combination of Karatsuba and the schoolbook methods is presented in Table 2. It can be observed that the combination of Karatsuba plus the schoolbook approaches outperforms the one reported by Scott for the range of word-lengths relevant to this work.

3) INTEGER SQUARING OPERATION

For operands with a word-length $n \geq 6$, we chose a variant of the Karatsuba method that takes advantage of the repeated partial products that show up in the squaring operation. Considering an n-word integer a written as $a = a_0 + a_1 \cdot r^{n/2}$. First compute the values $c_L = a_0^2$, $c_M = 2(a_0 \cdot a_1)$ and $c_H = a_1^2$. Then the squaring $c = a^2$ can be computed as $c = c_L + 2(a_0 \cdot a_1) \cdot r^{n/2} + c_H$. This can be obtained at the cost of two $n/2$-word squarings, one multiplication of $n/2$-word operands, and two n-word additions.

The implementation of an n-word squaring for $n \in \{24, 16, 8\}$ was conducted using up to three Karatsuba levels (going from 24- to 12-, then to 6-word and finally to 3-word multiplications/squarings). Eighteen squaring and nine multiplications of 3-words operands were computed for $n = 24$. For $n = 16$, two Karatsuba levels (going from 16- to 8- and then to 4-word multiplications/squarings), require six squarings and three multiplications of 4-word operands. For $n = 8$ it suffices one Karatsuba level (going from 8- to 4-word multiplications/squarings), using two squarings and one multiplication of 4-word operands. According to our experiments, the squaring computation of up to 4-word operands has a better performance when using a pure schoolbook approach. In the case of operands with a word-length $n \geq 6$, the best strategy was to combine the Karatsuba and schoolbook methods. A summary of the results obtained are reported in Table 3.

TABLE 3. A comparison of integer squaring using the schoolbook and the Karatsuba methods. Computational costs are reported in number of word multiplications (using `MUX` instructions) and clock cycles measured on Haswell (HW) and Skylake (SK) micro-architectures.

n	Word muls		Clock cycles			
	Karatsuba	schoolbook	Karatsuba		schoolbook	
			HW	SK	HW	SK
2	3	3	8	6	8	6
3	6	6	20	14	20	14
4	10	10	29	26	28	23
6	21	21	60	55	60	51
8	36	-	123	109	-	-
12	57	-	274	194	-	-
16	100	-	511	331	-	-
24	235	-	1024	713	-	-

4) MONTGOMERY MODULAR REDUCTION

The Montgomery modular reduction of Equation (1), requires to compute two n-word multiplications, which are divided or reduced modulo $R = r^n$. A straightforward optimization can be applied observing that for the multiplication $\mu \cdot d \mod R$, only the least significant half of the result must be computed. Likewise, in the case of the $(\mu \cdot d \mod R) \cdot p$ computation, only the most significant half of the product is required (due to the subsequent division by R).

For the cases when $n \leq 8$, these operations were performed using the schoolbook multiplication method. Thus, an n-word multiplication divided by R is computed using $n(n+1)/2+n$ word multiplications; and an n-word multiplication modulo R is computed using $n(n+1)/2$ word multiplications. On the other hand, for the cases when $n > 8$ we employed up to two levels of the Karatsuba method. At each level was necessary to compute one $n/2$-word multiplication and two half $n/2$-word multiplications as depicted in Figure 1.

Table 4 reports the timings measured for the Montgomery modular reduction, modular multiplication and squaring operations. The operands have a word length of $n \in \{8, 16, 24\}$.

5) MONTGOMERY-BASED RSA SIGNATURE

The RSA signature described in the introduction section, was performed using the exponentiation procedure presented in Algorithm 2, which was implemented with an underlying Montgomery-based arithmetic layer. Furthermore, the two RSA signature exponentiations were computed concurrently using two cores and the OpenMP library for synchronization.

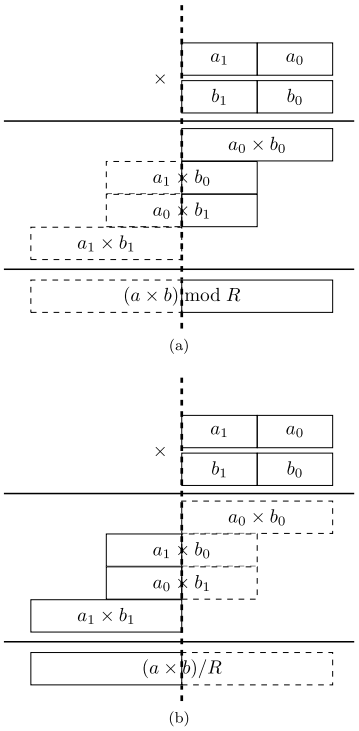

FIGURE 1. Let a and b be two n-word integers written as $a = a_0 + a_1 \cdot r^{n/2}$ and $b = b_0 + b_1 \cdot r^{n/2}$, respectively. Figure (a) shows a Karatsuba n-word multiplication modulo R, whereas Figure (b) shows a Karatsuba n-word multiplication divided by R. The dashed rectangles show the operations that are not computed.

TABLE 4. Timings for modular reduction, modular multiplication and modular squaring. All timings are reported in clock cycles measured on Haswell (HW) and Skylake (SK) micro-architectures.

Algorithm	Clock cycles					
	8-word		16-word		24-word	
	HW	SK	HW	SK	HW	SK
Montgomery reduction	232	224	900	728	1864	1582
Modular multiplication	424	349	1628	1233	3132	2688
Modular squaring	420	338	1500	1131	2820	2395

The 8-word modular exponentiations for RSA-1024 were performed using a window size $\omega = 4$. In the case of the 16- and 24-word modular exponentiations (required for computing RSA-2048 and RSA-3072), a window size $\omega = 5$ was chosen.

In Table 5, the latency achieved by our library when computing RSA signatures for 1024- , 2048-, and 3072-bit private keys is reported. Table 5 presents a comparison of our results against previously reported RSA implementations. Bos et al. *et al.* in [11] computed a Montgomery multiplication by splitting the Montgomery algorithm into two parts, which can be executed in parallel using Single Input Multiple Data (SIMD) instructions. Moreover, the authors of [11] presented RSA signature timings corresponding to a serial implementation. Table 5 also shows the work by Gueron and Krasnov [12], where the authors reported an RSA implementation that profited from a redundant integer representation that avoids the carry propagation by using operands organized

TABLE 5. Performance comparison of the RSA signature implemented on CPU platforms using Montgomery based arithmetic. All timings are given in millions of clock cycles.

Work	Clock cycles (Millions)			P
	RSA-1024	RSA-2048	RSA-3072	
Bos [11][1] (SSE)	2.09	12.21	-	✗
Bos [11][1]	0.88	4.92	-	✗
Gueron [12][2]	-	2.1	9.6	✗
Gueron [12][3]	-	1.9	6.0	✗
This work[2]	**0.29**	**1.92**	**5.93**	✓
This work[3]	**0.25**	**1.65**	**5.02**	✓

[1] Sandy Bridge, [2] Haswell, [3] Skylake
P = Protected Implementation.

on 29-bit words. To the best of our knowledge, none of these two works included countermeasures to protect their RSA implementations against some basic side-channel attacks.

B. CPU IMPLEMENTATION OF RNS-BASED ARITHMETIC

In this section, our implementation of RNS-based arithmetic using the set of Advanced Vector Extensions 2 AVX2 is presented. A performance comparison between the RNS reduction procedures given in Algorithm 3 and Algorithm 4 is also reported.

1) VECTOR INSTRUCTIONS

In order to perform an efficient implementation of the RNS based arithmetic as described in Section §II-C, we took advantage of the AVX2 instruction set introduced in the Intel Haswell micro-architecture [31]. AVX2 is an extension from its ancestor AVX, which allows to compute Single Instruction Multiple Data (SIMD) operations using 256-bit vector registers. AVX2 provides operations supporting integer arithmetic that are able to compute up to four concurrent 64-bit operations over the values stored in the vector registers. In terms of performance, one would expect a speedup factor of four coming from the simultaneous execution of 64-bit operations. Nevertheless, this acceleration can be attained only by some instructions. This is due to some overhead factors like the execution latency and throughput, and the number of execution units available in the target micro-architecture. In fact, the size of the AVX2 multiplier is expected to be a limiting factor. We mainly made use of the following AVX2 instructions,

- **`mm256_mul_epu32`**: Computes four products of 32×32 bits, storing the four 64-bit results on a 256-bit vector register;
- **`mm256_add_epi32, mm256_sub_epi32`**: Computes eight concurrent 32-bit additions/subtractions, without handling the input/output carry and borrow, respectively;
- **`mm256_slli_epi32,mm256_srli_epi32`**: Computes eight 32-bit logical shifts using the same fixed shift displacement for every word stored in the vector register;
- **`mm256_shuffle_epi32`**: Shuffles 32-bit values of the source vector in the destination vector at the locations selected by a control operand;

- **mm256_xor_si256, mm256_and_si256**: Computes the XOR/AND of two 256-bit vector registers;
- **mm256_cmpgt_epi32**: Returns a vector with the values $2^{32}-1$ and zero depending if the comparison of the 32-bit integers in the vector register is true or not.

Since the AVX2 multiplier works on 32-bit input operands, a word size $w = 32$ must be assumed. This implies that the operations required for RSA signatures with 1024-, 2048- and 3072-bit keys must be computed using integers with a word-length $n \in \{16, 32, 48\}$.

RNS addition, subtraction, and multiplication of two integers a and b are performed component-wise as shown in Equation (3) of Section §II-C. Considering our target micro-architectures, one can compute up to eight operations modulo m_i simultaneously. Therefore, all the computations described below must be performed by each vector storing the moduli of the the RNS-basis $\mathcal{B}$, *i.e.* a total of $\lceil \frac{n}{8} \rceil$ vectors.

The implementation of the RNS arithmetic main operations taking advantage of the AVX2 instructions is presented in the remaining of this section.

2) VECTORIAL RNS ADDITION/SUBTRACTION

Addition and subtraction can be straightforwardly implemented using the vector operations included in the AVX2 instruction set. Initially, the integer addition or subtraction are computed with the mm256_add_epi32 or mm256_sub_epi32 instructions. The result of these operations is stored in a vector C. As shown in Figure 2, the modular reduction by each moduli m_i belonging to the RNS base $\mathcal{B}$ (cf. Section §II-C) can be computed in constant time as discussed next.

By means of the mm256_cmpgt_epi32 instruction, one can catch the carry or borrow produced by the integer addition or subtraction operations, which is stored in a vector CB. Thereafter, the instruction mm256_and_si256 is used to compute the logic AND of CB and the vector $\mathcal{M}$ of moduli m_i, whose result is stored in a vector D. Then, the vector D is subtracted or added to the value obtained from the above addition or subtraction, respectively. The RNS computation of $C = A \oplus B$ and $C = A \ominus B$ is depicted in Figure 2.

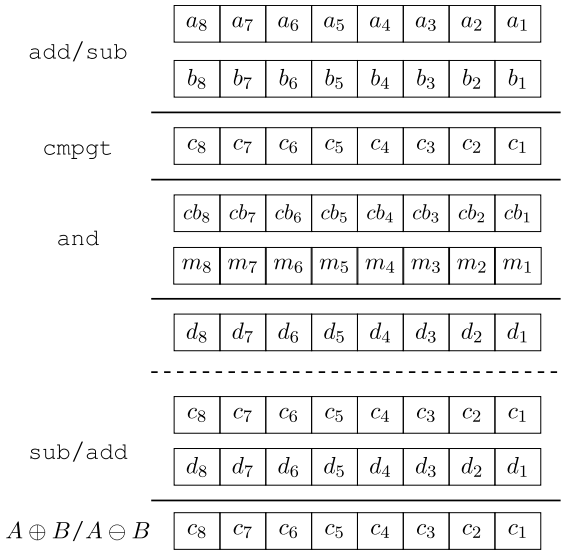

FIGURE 2. RNS addition/subtraction using AVX2 instructions.

3) VECTORIAL RNS MULTIPLICATION

Multiplication and squaring in RNS are a bit more involved operations than the addition and subtraction ones. This is because, the AVX2 instruction mm256_mul_epu32 only computes four 32×32-bit multiplications. Hence, in order to compute a component-wise integer multiplication of two RNS vectors A and B, we use the mm256_mul_epu32 instruction. This instruction calculates the products of odd indexes and store them into a vector D_0. Then, the shuffle instruction mm256_shuffle_epi32 can be used to reorder the 32-bit values of the A and B vector registers. This permits to compute the products of the even indexes, which are stored in the vector D_1 as shown in Figure 3.

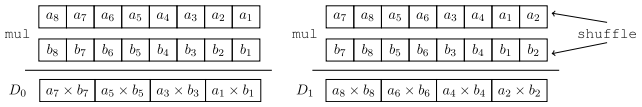

FIGURE 3. Component-wise integer multiplication of two integers a and b in RNS representation.

RNS INDIVIDUAL MODULAR REDUCTION

Modular reduction by each moduli $m_i = 2^w - \mu_i$ in $\mathcal{B}$ is computed as described in Remark 2 of Section §II-C. Let D_0 and D_1 be two output vectors of the computation shown in Figure 3, and let $\mathcal{M}$ be a vector composed by the μ_i small values described in Remark 2.

First, the mm256_shuffle_epi32 instruction is executed to reorder the 32-bit values in the D_0, D_1 and $\mathcal{M}$ vectors. Thereafter, the instruction mm256_mul_epu32 recovers the values $\mu_i \cdot \lfloor t_i/2^w \rfloor$ with $t_i = a_i \cdot b_i$, which were stored in the vectors E_0 and E_1, respectively. Then, the execution of the mm256_srli_epi32 instruction on E_0 and E_1 using an offset of 32 produces the vectors F_0 and F_1, which are added using mm256_add_epi64 to D_0 and D_1. This obtains the values $d_i = t_i \bmod 2^w + \mu_i \cdot \lfloor t_i/2^w \rfloor$. After two iterations of the above procedure, the d_i values stored in D_0 and D_1 correspond to $t_i \bmod m_i$. Thus, it becomes necessary to combine the final vectors D_0 and D_1 to get the vector D that stores the values of $A \otimes B$. This procedure is depicted in Figure 4.

4) VECTORIAL RNS MODULAR REDUCTION

Modular reduction was performed using Algorithm 3 as was presented by Jeljeli in [21], and Algorithm 4 as was described by Kawamura in [24]. For both of these two reduction algorithms it becomes necessary to find the approximation $\hat{\alpha}$ as given in Equation (7), which was computed as described in Section §II-C, Remark 3.

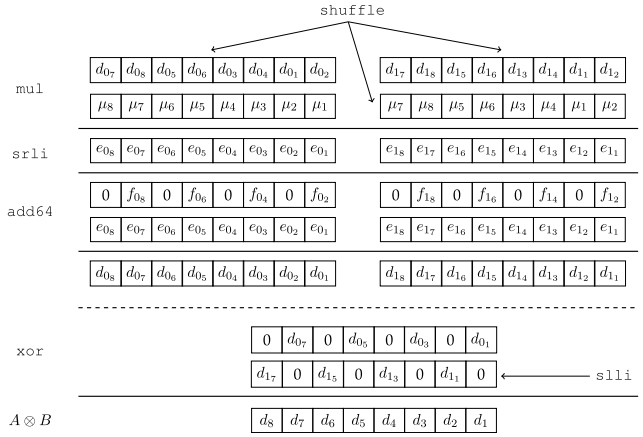

FIGURE 4. RNS multiplication/squaring using AVX2 instructions.

TABLE 6. Comparison of timings for modular reduction, modular multiplication and modular squaring based on the RNS reduction Algorithm 3 and Algorithm 4 using the AVX2 instructions. All timings are reported in clock cycles measured on Haswell (HW) and Skylake (SK) micro-architectures.

Algorithm	Clock cycles					
	8-word		16-word		24-word	
	HW	SK	HW	SK	HW	SK
Algorithm 3	3,322	2,943	11,862	10,059	26,046	22,420
Modular mult	3,522	3,039	12,066	10,332	27,450	22,627
Modular sqr	3,402	3,008	12,050	10,270	26.314	22,589
Algorithm 4	1,434	1,330	4,902	4,012	12,042	10,571
Modular mult	1,498	1,385	5,074	4,131	12,350	10,776
Modular sqr	1,494	1,382	5,066	4,123	12,206	10,750

When working with the RNS reduction Algorithm 3, one computes $\hat{\alpha}$ for each vector storing $\Gamma = (\gamma_1, \ldots, \gamma_\ell)$. This calculation is performed by invoking the `mm256_srli_epi32` instruction with offsets 5 for RSA-1024, and 25 for RSA-2048 and RSA-3072. For the reduction Algorithm 4, offsets of 18 for RSA-1024 and RSA-2048, and 16 for RSA-3072 were employed.

Thereafter, the `mm256_slli_epi32` instruction is applied to each vector using offsets that guarantee constraining the subsequent additions in the interval $[0, 2^{32} - 1]$. For example when the reduction Algorithm 3 is used, the offsets are 19 for RSA-1024, and 17 for RSA-2048 and RSA-3072. In the case of the reduction Algorithm 4, the offsets are 14, 10 and 8 for RSA-1024, RSA-2048 and RSA-3072, respectively. As a final step, all vectors are added using `mm256_add_epi32` instructions, and the values of the resulting output vector are also added in order to obtain a 32-bit value, which is shifted to the right by an offset of 24.

The matrix-vector multiplications needed in both algorithms can be performed using ℓ RNS multiplications followed by $\ell - 1$ RNS additions, as shown in Section §II-C. The matrix multiplication in Step 9 of Algorithm 3 can be done straightforwardly. However, the matrix multiplications in Steps 10 and 17 of Algorithm 4 require to transpose the matrices $|M_i|_{m'_j}$ and $\left|M'_i\right|_{m_j}$.

TABLE 7. Timings for RSA signature algorithm using AVX2 instructions. All timings are reported in millions of clock cycles measured on Haswell (HW) and Skylake (SK) micro-architectures.

RNS reduction algorithm	Clock cycles (Millions)						P
	RSA-1024		RSA-2048		RSA-3072		
	HW	SK	HW	SK	HW	SK	
Jeljeli [21]	2.3	2.0	15.1	12.9	48.7	41.9	✓
Montgomery [24]	0.99	0.90	6.3	5.1	22.5	19.8	✓

P = Protected Implementation.

The experimental results obtained for both reduction algorithms are presented in Table 6. One can observe that the RNS Montgomery reduction of Algorithm 4 is twice as fast as the RNS reduction Algorithm 3. This is mainly due to the fact that for the RNS Montgomery reduction the basis used to represent the numbers are of size $\ell = n$, whereas the base used in Algorithm 3 has a size of $\ell = 2n + 3$. Moreover, all the operations in Algorithm 4 require to process vectors with a size of roughly half of the ones required in Algorithm 3.

5) RNS-BASED RSA SIGNATURE

As in §III-A, we concurrently computed two RSA modular exponentiations using two cores running the protected exponentiation method described in Algorithm 2. Once again, the synchronization of these two tasks was achieved trough the usage of the OpenMP library. Modular exponentiations for RSA-1024 were performed using a window size $\omega = 4$, whereas $\omega = 5$ was adopted for modular exponentiations corresponding to both RSA-2048 and RSA-3072.

Table 7 presents the latency achieved by our library for the RSA-1024, RSA-2048, and RSA-3072 signature computations. One can observe that the RSA signature based on RNS Montgomery arithmetic (Algorithm 4) is two times faster than the RSA signature based on the RNS reduction Algorithm 3. It is worth noting that the best results of Table 7 are slower by a factor of 3.1x and 3.8x than the best results reported in Table 5 for RSA-2048 and RSA-3072. On the other hand, the best result for RSA-1024 signature in Table 7 is 2.5 times faster than the one reported in Table 5.

IV. EFFICIENT IMPLEMENTATION OF RSA ON GPU PLATFORMS

In this section, the implementation of the RSA exponentiation on a GPU architecture is described. The material presented here is partially based on a previous work presented in [32].

A. PARALLEL COMPUTATIONS ON GPU ARCHITECTURES

Graphics Processing Units (GPU) are optimized hardware blocks originally designed for performing graphics operations [33]. Nowadays, GPU platforms are widely considered general purpose processors. In 2006, NVIDIA introduced a parallel computing framework named `CUDA`, which was especially designed for GPU environments. `CUDA` defines three important features: a threading model, a set of conventions for calling native GPU's functions, and a hierarchical memory infrastructure. In a GPU architecture the basic computational and resource allocation units are *threads*. Threads can be

grouped into *blocks*, which in turn can be grouped into a *grid*. Threads in a block are partitioned into warps. For all GPU architectures a warp is composed by 32 threads that run concurrently.

A GPU architecture utilizes the Single Instruction Multiple Thread (SIMT) programming model paradigm, where all threads inside a warp can execute the same instruction at the same time. The general programming model consists of code sequences called kernels. A kernel execution can be synchronous or asynchronous. This allows programmers to manage concurrent execution through the completion of command sequences called streams.

GPU architectures support several types of memory models such as, global memory, constant memory, shared memory, among others. The shared memory is a small cache memory with low-latency attached to each Streaming Multiprocessor (SM). Shared memory can be accessed by all the threads in a block. During kernel invocation, a programmer can configure the amount of shared memory available per block.[4] For example, in a Kepler architecture a valid configuration can allocate 48 KB and 16KB for the software and the hardware data cache, respectively. Moreover, the `PTX` (Parallel Thread eXecution) is a low-level parallel thread execution virtual machine that provides a stable programming model and an instruction set for general purpose parallel programming [34]. It is often used to gain control over arithmetic operations trying to avoid thread *divergence* during a program execution.[5]

In this work we made extensive use of the following assembly instructions,

- **`addc`**: Adds two 32/64-bits values taking into account the carry-in bit, producing a carry-out bit;
- **`subc`**: Performs a 32/64-bits subtraction operation with input borrow and producing a borrow-out bit;
- **`mul.lo`**: Multiplies two 32/64-bits values and returns $x_i \times y_i \bmod 2^r$, where x_i and y_i are both non-negative integers, and r is typically selected to be the GPU word size;
- **`mul.hi`** : Multiplies two 32/64-bits values and returns $x_i \times y_i / 2^r$, where x_i and y_i are both non-negative integers;
- **`mad.(hi,lo).cc`**: Multiplies two 32/64 -bits values, extracts the higher or lower half of the result, and adds a third 32/64-bit value with carry-out.

Especially because of its ability to handle an implicit carry/borrow operation, the aforementioned instructions helped our implementation to achieve a better performance. Also, we extensively used the data type *uint2*, which is a two-element vector that stores two halves of a 64-bit integer, as the most and least significant halves. This data structure permits an efficient access to data stored in registers and the shared memory.

[4]Common GPU architectures are Fermi, Kepler, Haswell, Pascal, and Volta. Each architecture has different sizes for shared memory.

[5]A thread divergence occurs when several threads do not execute the same instruction at the same time. This prevents to fully exploit parallelism.

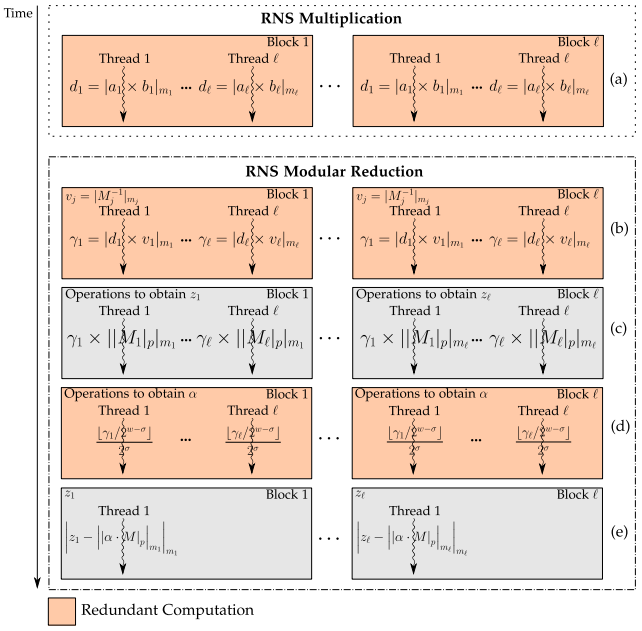

FIGURE 5. Computation of RNS modular multiplication on a GPU platform.

B. MAIN OPERATIONS IN RNS REPRESENTATION

In the following, we describe how the RSA exponentiation was carried out in the GPU platform. As a pre-computation step, in the CPU server the set of pair-wised co-prime numbers composing the RNS-basis $\mathcal{B}$ was chosen. Then, all the RSA operands and moduli were converted to their RNS representation and these values were sent to the GPU platform. After that, the exponentiation computation in the GPU platform was considered ready to start.

RNS INTEGER MULTIPLICATION

Integer multiplication can be performed in parallel by launching ℓ blocks with ℓ threads, which compute concurrently up to ℓ independent RNS multiplications of the form $C = A \otimes B$, where A and B are in RNS representation in base $\mathcal{B} = m_1, \cdots, m_\ell$ (or in base $\mathcal{B} = m_1, \cdots, m_\ell$ and $\mathcal{B}' = m'_1, \cdots, m'_\ell$ if the RNS reduction Algorithm 4 is used). This arrangement is depicted in Figure 5(a), where it is shown that each thread is in charge of processing the modular product of a pair of RNS coordinates $|a_i \cdot b_i|_{m_i}$ (or $|a_i \cdot b_i|_{m_i}$ and $|a_i \cdot b_i|_{m'_i}$). Since each warp executes the same instruction, this arrangement avoids thread's divergence. Moreover, the multiplications carried out concurrently do not need to be synchronized. Also, the threads can efficiently access each coordinate of the RNS vectors as these values are allocated on contiguous segments of memory. Each thread stores the output of its modular multiplication computation on a register, thus avoiding global memory accesses that would be much more costlier.

After all threads have completed the integer multiplication step, a modular reduction by the modulus p must be applied either using the reduction Algorithm 3 or the RNS Montgomery reduction algorithm 4. For the sake of brevity,

TABLE 8. Performance comparison of RSA private operation implemented in GPU platforms. Al timings are given in milliseconds.

Work	GPUs		Latency (ms)			PI*
	Model	@ GHz	RSA-1024	RSA-2048	RSA-3072	
Jang *et al.* [9]	GTX 580	1.54	3.8	13.8	-	✗
Fadhil *et al.* [8]	GT 750	0.8	2.8	17.2	50.1	✗
Yang *et al.* [10]	GT 750	0.96	2.6	6.5	-	✗
Dong *et al.* [7]	GTX TITAN	0.88	-	10.8	26.6	✗
This work[1]	GTX TITAN	0.88	**2.1**	**5.1**	**8.9**	✓
This work[1]	GTX 1080	1.81	**1.0**	**2.1**	**3.4**	✓
This work[2]	GTX 1080	1.81	**2.3**	**5.0**	**8.2**	✓

*Protected Implementation.
[1]Modular Reduction using Algorithm 3
[2]Modular Reduction using Algorithm 4

we only explain in the following our GPU implementation of the RNS reduction Algorithm 3. The corresponding implementation of the RNS Montgomery reduction algorithm 4 follows a similar design flow.

C. RNS MODULAR REDUCTION USING ALGORITHM 3

Modular reduction carried out by Algorithm 3 is illustrated in Figures 5 (b), (c), (d), and (e). The reduction process requires the pre-computation of several values (Steps 1-3), which are processed in the hosting CPU and sent to the GPU before the main computation starts. The RNS vector $|M_i^{-1}|_{m_i}$ and the RNS table $|M_i|_p$ in Steps 1-2 are both stored in the shared memory so that it can be available for all the threads. The third precomputed value is the table containing the RNS vectors $|\alpha \cdot M|_p$, for $\alpha = 1, \ldots, \ell - 1$. This table is mapped to the texture memory because only few threads have to query it.

The multiplication operations required in Step 5 are computed in a redundant fashion as previously described and illustrated in Fig. 5b. Then in Step 9, the most expensive task of the reduction algorithm is performed, requiring the computation of ℓ and $\ell-1$ RNS multiplications and additions, respectively. This calculation is performed in parallel by launching ℓ blocks with ℓ threads each (illustrated in Fig. 5c). If there are more than 32 active threads, then an explicit barrier must be placed in order to synchronize all threads of each block, and one must wait until all the threads have completed their execution. Once that all the partial results have been obtained by each block, each thread stores its result in the shared memory. Next, all the partial results so obtained must be added, This can be done by using a binary addition tree strategy [35]. Step 10 of Algorithm 3 calculates ℓ copies of α using ℓ blocks as shown in Figure 5(d). The $\ell - 1$ additions are computed collaboratively as previously mentioned. Finally, in Step 11 of Algorithm 3, a single thread of each one of the ℓ blocks, performs an RNS coordinate subtraction saving the final result of the modular reduction into the global memory (see Figure 5e). This avoids that the threads compete to each other for writing into the same memory address.

D. GPU RESULTS

The experimental hardware setup used for the experimental results reported in this section is the following: CUDA toolkit version 9.1, 20 MultiProcessor with 128 cores each running at 1.81 GHz, and global memory of 8 GB.

In Table 8, the latency achieved by our GPU library for the RSA private operation is reported for key lengths of 1024, 2048 and 3072 bits, respectively. Table 8 also shows a comparison against related works previously published for the parallel RSA implementation on GPU platforms. It can be seen that our implementation has better latency for RSA-1024, RSA-2048 and RSA-3072 than previously published results. Besides, our implementation offers a first-line of protection against timing attacks.

V. CONCLUSION

In this paper an optimized parallel implementation of RSA signatures using some of the most efficient and effective arithmetic algorithms for both CPU and GPU high-end architectures was presented. As it was shown in Tables 5 and 8, our RSA CPU and GPU libraries instantiated with the most popular private key lengths of 1024, 2048 and 3072 bits, achieve faster timings than previously reported literature.

From an algorithmic point of view, it is also interesting to compare the performance achieved by the RNS reduction procedures described in Algorithms 3-4 when implemented in CPU and GPU architectures. In the case of our RSA CPU implementation, a close inspection of Table 7 reveals that due to the higher number of about $4n^2$ word multiplications required by Algorithm 3 compared with about $2n^2$ word multiplications required by Algorithm 4, the latter reduction algorithm is faster than the former by a factor close to two. Strikingly, from our RSA GPU implementation we observed the opposite situation. Indeed, due to the fact that Algorithm 3 is more amenable for the massive parallelism provided by the GPU many-core architecture, it yields a better performance than Algorithm 4. This computational behavior is reported in Table 8.

In spite of its massive parallelism, we observe that GPU implementations of RSA are considerably slower than their CPU counterparts. Nevertheless, notice that our RSA GPU implementation enjoys a sub-quadratic complexity in the cost of the RSA exponentiation with respect to the size of its key.

For example, from our GPU timings shown in Table 8, one can see that the computational timing cost of RSA-2048 and RSA-3072 is just 2.42 and 4.23 more expensive than RSA-1024, respectively. On the contrary, from our CPU timings shown in Table 5, one can see that the cost of RSA-2048 and RSA-3072 is 6.60 and 20.08 more expensive than RSA-1024, respectively. These timings increments closely follow a quadratic complexity. Hence, we believe that for cryptographic applications where extremely large operands are required (such as the ones proposed in several homomorphic encryption schemes), our RNS arithmetic library could be of interest. We leave as a future work to study this potential application.

REFERENCES

[1] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 26, no. 1, pp. 96–99, Jan. 1983.

[2] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Advances in Cryptology—CRYPTO* (Lecture Notes in Computer Science). London, U.K.: Springer-Verlag, 1996, pp. 104–113. [Online]. Available: http://dl.acm.org/citation.cfm?id=646761.706156
[3] E. Barker, "Recommendation for key management part 1: General," NIST, Gaithersburg, MD, USA, Tech. Rep. NIST Special Publication 800-57 Part 1 Revision 4, Jan. 2016. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf
[4] D. Adrian, L. Valenta, B. Vandersloot, E. Wustrow, S. Zanella-Béguelin, P. Zimmermann, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, and E. Thomé, "Imperfect forward secrecy: How Diffie-Hellman fails in practice," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2015, pp. 5–17.
[5] M. Delcourt, T. Kleinjung, A. Lenstra, S. Nath, D. Page, and N. Smart, "Using the cloud to determine key strengths—Triennial update," Cryptol. ePrint Arch., Tech. Rep. 2018/1221, 2018. [Online]. Available: https://eprint.iacr.org
[6] J.-S. Coron, "On the exact security of full domain hash," in *Proc. 20th Annu. Int. Cryptol. Conf.*, Santa Barbara, CA, USA, Aug. 2000, pp. 229–235.
[7] J. Dong, F. Zheng, W. Pan, J. Lin, J. Jing, and Y. Zhao, "Utilizing the double–precision floating–point computing power of GPUs for RSA acceleration," *Secur. Commun. Netw.*, vol. 2017, pp. 1–15, Sep. 2017.
[8] H. Mohammedfadhil and M. I. Younis, "Parallelizing RSA algorithm on multicore CPU and GPU," *Int. J. Comput. Appl.*, vol. 87, no. 6, pp. 15–22, Feb. 2014.
[9] K. Jang, S. Han, S. Han, S. Moon, and K. Park, "SSLShader: Cheap SSL acceleration with commodity processors," in *Proc. 8th USENIX Conf. Netw. Syst. Design Implement.*, in NSDI. Berkeley, CA, USA: USENIX Association, 2011, pp. 1–14.
[10] Y. Yang, Z. Guan, H. Sun, and Z. Chen, "Accelerating RSA with fine-grained parallelism using GPU," in *Proc. 11th Int. Conf. Inf. Secur. Pract. Exper.*, in ISPEC. Beijing, China: Springer, 2015, pp. 454–468.
[11] J. W. Bos, P. L. Montgomery, D. Shumow, and G. M. Zaverucha, "Montgomery multiplication using vector instructions," in *Proc. 20th Int. Conf. Sel. Areas Cryptogr.*, Burnaby, BC, Canada, Aug. 2013, pp. 471–489, doi: 10.1007/978-3-662-43414-7_24.
[12] S. Gueron and V. Krasnov, "Speed records for multi-prime RSA using AVX2 architectures," in *Proc. 13th Int. Conf. Inf. Technol., New Gener.* Cham, Switzerland: Springer, 2016, pp. 237–245, doi: 10.1007/978-3-319-32467-8_22.
[13] M. Joye and M. Tunstall, "Exponent recoding and regular exponentiation algorithms," in *Proc. 2nd Int. Conf. Cryptol. Africa*, in AFRICACRYPT. Gammarth, Tunisia: Springer, 2009, pp. 334–349.
[14] T. Oliveira, J. López, and F. Rodríguez-Henríquez, "Software implementation of Koblitz curves over quadratic fields," in *Cryptographic Hardware and Embedded Systems—CHES*, B. Gierlichs and A. Y. Poschmann, Eds. Berlin, Germany: Springer, 2016, pp. 259–279.
[15] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comp.*, vol. 44, no. 170, p. 519, May 1985.
[16] D. J. Bernstein, "Multidigit modular multiplication with the Explicit Chinese Remainder Theorem," Ph.D. dissertation, Univ. California, Berkeley, Berkeley, CA, USA, 1995, ch. 4.
[17] J. M. Couveignes, "Computing a square root for the number field sieve," in *The Development of the Number Field Sieve* (Lecture Notes in Mathematics), vol. 1554, H. W. Lenstra and A. K. Lenstra, Eds. Berlin, Germany: Springer-Verlag, 1993, pp. 90–97.
[18] P. L. Montgomery and R. D. Silverman, "An FFT extension to the P—1 factoring algorithm," *Math. Comput.*, vol. 54, no. 190, p. 839, Apr. 1990.
[19] P. L. Montgomery, "An FFT extension of the elliptic curve method of factorization," Ph.D. dissertation, Univ. California, Los Angeles, CA, USA, 1992.
[20] J.-C. Bajard, J. Eynard, and N. Merkiche, "Montgomery reduction within the context of residue number system arithmetic," *J. Cryptograph. Eng.*, vol. 8, no. 3, pp. 189–200, Sep. 2018.
[21] H. Jeljeli, "Accelerating iterative SpMV for discrete logarithm problem using GPUs," in *Proc. 5th Int. Workshop Arithmetic Finite Fields (WAIFI)*, in Lecture Notes in Computer Science, vol. 9061, Ç. K. Koç, S. Mesnager, and E. Savaş, Eds. Gebze, Turkey, 2014.
[22] H. Jeljeli, "Accélérateurs logiciels et matériels pour l'algèbre linéaire creuse sur les corps finis," LORIA—ALGO—Dept. Algorithms, Comput., Image Geometry, Inria Nancy—Grand Est, Villers-Lès-Nancy, France, 2015. [Online]. Available: https://hal.inria.fr/tel-01178931
[23] K. Posch and R. Posch, "Modulo reduction in residue number systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, no. 5, pp. 449–454, May 1995.
[24] S. Kawamura, Y. Komano, H. Shimizu, and T. Yonemura, "RNS montgomery reduction algorithms using quadratic residuosity," *J. Cryptograph. Eng.*, vol. 9, no. 4, pp. 313–331, Nov. 2019, doi: 10.1007/s13389-018-0195-8.
[25] S. Kawamura, M. Koike, F. Sano, and A. Shimbo, "Cox-rower architecture for fast parallel montgomery multiplication," in *Advances in Cryptology—EUROCRYPT*, B. Preneel, Ed. Berlin, Germany: Springer, 2000, pp. 523–538.
[26] C. Walter, "Montgomery exponentiation needs no final subtractions," *Electron. Lett.*, vol. 35, no. 21, p. 1831, Oct. 1999.
[27] I-Corporation. (Nov. 2018). *Intel 64 and IA-32 Architectures Optimization Reference Manual*. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf
[28] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," *Sov. Phys. Doklady*, vol. 7, no. 7, pp. 595–596, 1963.
[29] M. Scott, "Missing a trick: Karatsuba variations," *Int. Assoc. Cryptologic Res.*, vol. 2015, p. 1247, Jan. 2016. [Online]. Available: http://eprint.iacr.org/2015/1247
[30] A. Weimerskirch and C. Paar, "Generalizations of the Karatsuba algorithm for efficient implementations," Cryptol. ePrint Arch., Tech. Rep. 2006/224, 2006. [Online]. Available: http://eprint.iacr.org/2006/224
[31] I-Corporation. (Nov. 2018). *Intel Advanced Vector Extensions Programming Reference*. [Online]. Available: https://software.intel.com/sites/default/files/4f/5b/36945
[32] N. Cruz-Cortés, E. Ochoa-Jiménez, L. Rivera-Zamarripa, and F. Rodríguez-Henríquez, "A GPU parallel implementation of the RSA private operation," in *High Performance Computing—CARLA* (Communications in Computer and Information Science), Mexico City, vol. 697, C. J. B. Hernández, I. Gitler, and J. Klapp, Eds. 2017, pp. 188–203.
[33] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general–purpose computation on graphics hardware," *Comput. Graph. Forum*, vol. 26, no. 1, pp. 80–113, Mar. 2007.
[34] nVidia. *Parallel Thread Execution ISA V5.0, Application Guide*. Accessed: Sep. 2016. [Online]. Available: http://docs.nvidia.com/cuda/pdf/ptx_isa_5.0.pdf
[35] M. Harris, *Optimizing Parallel Reduction in CUDA*, document, nVidia, Santa Clara, CA, USA, 2008. [Online]. Available: https://arcb.csc.ncsu.edu/~mueller/mpigpu/readings/cuda_reduction.pdf

EDUARDO OCHOA-JIMÉNEZ received the B.Sc. degree in computer engineering from Metropolitan Autonomous University (UAM), Mexico, in 2010, and the M.Sc. and Ph.D. degrees in computer science from CINVESTAV, Mexico, in 2013 and 2019, respectively. His major research interests are in cryptography, finite field arithmetic, and software efficient implementation.

LUIS RIVERA-ZAMARRIPA received the B.Sc. degree in computer engineering from the Instituto Tecnológico de Ciudad Madero (ITCM), México, in 2006, and the M.Sc. and Ph.D. degrees in computer science from the Centro de Investigación en Computación (CIC), Instituto Politécnico Nacional (IPN), México, in 2012 and 2019, respectively. His major research interests are high-performance computing and finite field arithmetic.

NARELI CRUZ-CORTÉS received the Ph.D. degree from CINVESTAV, Mexico, in 2004. She is currently a Researcher Professor with the Centro de Investigación en Computación, Instituto Politécnico Nacional, Mexico City, Mexico. Her main research interests are cybersecurity, machine learning, and their applications.

FRANCISCO RODRÍGUEZ-HENRÍQUEZ is currently a Professor with the Department of Computer Science, CINVESTAV-IPN, Mexico City, where he joined, in 2002. He is a coauthor of *Cryptographic Algorithms on Reconfigurable Hardware*. His major research interests are in cryptography and finite field arithmetic.