Noname manuscript No.
(will be inserted by the editor)

Rethinking Modular Multi-Exponentiation in Real-World Applications

Vidal Attias · Luigi Vigneri · Vassil Dimitrov

Received: date / Accepted: date

Abstract The importance of efficient multi-exponentiation algorithms in a large spectrum of cryptographic applications continues to grow. Many of the algorithms proposed in the past pay attention exclusively on the minimization of the number of modular multiplications. However, a short reduction of the multiplicative complexity can be easily overshadowed by other figures of merit. In this article we demonstrate a large number of practical results aimed at concrete cryptographic tasks requiring multi-exponentiations and provide recommendations on the best possible algorithmic strategies for different selection of security parameters.

Keywords Multi-exponentiation · OpenSSL · benchmarking · cryptography · arithmetic

Mathematics Subject Classification (2020) MSC code1 · MSC code2 · more

V. Attias
IOTA Foundation
Berlin
Germany
E-mail: vidal.attias@iota.org

L. Vigneri
IOTA Foundation
Berlin
Germany
E-mail: luigi.vigneri@iota.org

V. Dimitrov
University of Calgary
Calgary
Canada
E-mail: vdimitro@ucalgary.ca

1 Introduction

1.1 Modular arithmetic

Modular arithmetic is a cornerstone of modern mathematics with innumerable applications in cryptography, coding theory, computer algebra and more. This field has been investigated since ancient Greece until today. For instance, the *modular reduction* problem

$$A \bmod N, \tag{MR}$$

which consists in obtaining the reminder of an integer A modulo another integer N, was already studied by Euclid, as written in the Elements [12]. During millennia, research produced highly sophisticated algorithms to compute (MR) such as the *Barrett reduction* [2], which computes the solution of (MR) by replacing divisions by multiplications, and the *Montgomery reduction* [18], which converts integers into a space where modulo operation is replaced by multiplications. The two algorithms solve the issue by replacing divisions by multiplications but the main difference between them is that Montgomery's conversion induces an overhead while its reduction operation *per se* is faster than Barrett's and it allows manipulated converted integers as if they were in a canonical form. Thus Montgomery is more interesting for computations involving lots of reductions using the same transformed integers whereas Barrett's will be privileged for one-time reductions. Both algorithms have been thoroughly studied both from a theoretical [6] and from an implementation perspective [14,7].

Another essential operation is the *modular exponentiation* defined as

$$x^e \bmod N, \tag{ME}$$

where x is an integer in $[0, N-1]$, and e is a natural number. Again, this problem received an important research coverage, both on purely theoretical aspects [10] and implementation perspective [4], with some research even focusing on producing quantum algorithms [21]. Sensible improvements in the solution of this problem has been given by the introduction of the *square-and-multiply* technique which uses the product of powers property, i.e., $m^p \cdot m^q = m^{p+q}$, and the binary representation of the exponent e as follows:

$$power(x, e) = \begin{cases} x, & \text{if } n = 1, \\ power(x^2, n/2), & \text{if } n \text{ is even}, \\ x \cdot power(x^2, (n-1)/2), & \text{if } n \text{ is odd}. \end{cases} \tag{1}$$

This technique led to a time complexity of $\mathcal{O}(\log(x))$ instead of $\mathcal{O}(x)$ and paved the way for a whole range of sophisticated algorithms using it [13].

A generalization of the problem in (ME) is called *modular multi-exponentiation*, and it is expressed as the product of n modular exponentiations:

$$\prod_{i=1}^{n} x_i^{e_i} \bmod N. \tag{MME}$$

Problem (MME) has been less profusely studied. One could naively compute each modular exponentiation separately and then compute their product but this has been shown to be suboptimal [15], hence the need for optimized algorithms.

1.2 Our contributions

A relevant metric traditionally used to compare the performance of multi-exponentiation algorithms is the total number of multiplications to be performed. Most (MME) algorithms aggregate the exponents during the scanning phase in order to reduce the amount of multiplications. This generally induces a precomputation overhead and adds some complexity resulting in a exponents filtering computation overhead. Although counting the number of modular multiplication is useful to compare algorithms regardless of the hardware, it does not catch side computations that can actually make a difference when it comes to implementation.

In this paper we tackle a practical problem: while it is easy to find extensive literature on the theoretical performance and guarantees of multi-exponentiation algorithms, little is known on how these techniques behave when implemented in real systems. The goal of the paper is indeed to provide the knowledge necessary to select the best algorithm depending on the hardware used and on the input parameters, such as modulus and base size. Furthermore, it is often non-trivial to tune the algorithm parameters, and we show that this is largely affected by the exponents and modulus size. In short, we found a lack of implementation study for multi-exponentiation algorithms, and the platform-dependant effects are not yet fully understood.

We present an extensive performance comparison of three multi-exponentiation algorithms operating on integers on single-core machines (in Section 3.6 we discuss about non-integers groups and multi-core machines). Specifically, we test different modern hardware: the Apple ARM M1 brand-new chip, a general-purpose Intel Core processor and the two latest generations of Raspberry Pis. Studying different hardware is important because platform-dependant side effects can occur. For example, the Raspberry Pi devices use a 32-bit operating system as opposed to modern general-purpose computers using 64 bits systems, which makes a difference for multi-precision computing libraries. Moreover, Raspberry Pis, which do not have efficient long division routines, make the optimization of the modular reduction an even more important task than for x86 devices. We reiterate that the contributions of this paper are twofold: (i) evaluate the performance differences depending on the hardware used to solve (MME); (ii) provide the reader a way to understand which multi-exponentiation algorithm to use and how to tune its parameters depending on application specifications.

The rest of the paper is organized as follows: in Section 2, we state the problem we are tackling. Then, in Section 3, we provide a description of the relevant algorithms used to solve (MME) along with pseudocode and a theoretical comparison. After that, Section 4 describes the basic concepts of the Montgomery reduction which will be an important component of our experiments. In Section 5, we show the results of these experiments. Finally, we conclude our paper in Section 6.

2 Problem statement

In this section, we formalize the problem which is addressed in the rest of the paper. First, consider r as the solution of the following (MME) problem:

$$r \equiv \prod_{i=1}^{n} x_i^{e_i} (\bmod N),$$

where $x_i \in \mathbb{N}$ with $i \in [1, n]$ represent the *bases* and have size λ bits, $N \in \mathbb{N}$ is the *modulus* and has size λ bits and e_i with $i \in [1, n]$ are the *exponents* and have size k bits. Furthermore, let $\Theta_{\lambda,k}(r,h,A)$ – or just $\Theta(r,h,A)$ for the sake of simplicity – be the computing time needed to solve r, where bases and modulus are of size λ and exponents are of size k, by algorithm A using hardware h. Hence, the goal of this paper is to provide insights to the following optimization problem:

$$\min_A \bar{\Theta}_{\lambda,k}(r,h,A), \qquad \forall \lambda, k \in \mathbb{N}, \tag{2}$$

where $\bar{\Theta}$ is the average computing time needed to solve r by algorithm A using hardware h. In other words, in this paper we provide extensive experimental results, supported by theoretical findings, in order to answer the following question: *For a given hardware, which multi-exponentiation algorithm provides the best performance depending on modulus and exponent sizes?*

Extensive studies exist on the number of multiplications needed to solve (MME) problems. We discuss some of the existing results in the next section, where we present the multi-exponentiation techniques which are used as a benchmark in this paper. Unlike prior work, we focus on the time spent computing the multi-exponentiation, rather than on the number of multiplications. Although the latter has its own merits, we show in this paper that using time does unveil unexpected behavior when choosing the best algorithm with respect to the hardware available and to the problem parameters.

3 Multi-exponentiation algorithms

In this section, we present the most relevant algorithms used to solve Problem (MME), among which we carefully analyze the implication of using separate exponentiations and (sliding) windowed algorithms. Each algorithm comes with different implementation complexities and may be optimal depending on the hardware considered or on the input parameters. In the rest of the section we present a description of the algorithms studied, including their detailed pseudocodes, and we provide a comparison based on their theoretical guarantees (we test empirically these results in a later section).

Important notation for pseudocodes

We can access the exponent's bits individually in the following way:

- $e_i[j]$ returns the j-th (starting with index 0) least significant bit of e_i.
- $e_i[a \dots b]$ returns the bits within the a-th and the b-th least significant bits.

If a or b are negative numbers, $|a|$ or $|b|$ zeros are added as least significant digits. For example, if $e_i = 10101011_2$, then $e_i[3 \dots 1] = 101_2$ and $e_i[3 \dots -2] = 101100_2$.

3.1 Separate exponentiations algorithm (SEPARATE)

The simplest way of computing (MME) is to perform separately the single exponentiations and then multiply them to get the result. Although being easy to understand and to implement, this method, which we will denote as SEPARATE, is nevertheless suboptimal as we show later. Algorithm 1 gives a pseudo-code of this algorithm.

The computation through SEPARATE method requires:

- k modular multiplications per exponentiation using square and multiply techniques as presented in Section 1.
- $n-1$ modular multiplications combining the previously computed results.

Algorithm 1: SEPARATE algorithm

```
  Data: Bases $\{x_i\}_{i\in[1,n]}$, exponents $\{e_i\}_{i\in[1,n]}$
1 and the modulus $N$;
  Result: $A = \prod_{i=1}^{n} x_i^{e_i}$
2 $A \leftarrow 1$;
3 for $i \leftarrow 1$ to $n$ do
4     $A \leftarrow A \cdot x_i^{e_i} (\bmod N)$;
5 return $A$
```

3.2 Windowed 2^w-ary algorithm (2^w-ARY)

A first optimisation consists of precomputing several combinations of products of the x_is so that we can use a technique similar to *square-and-multiply* with the n bases. This optimisation has been extended to computing a window of size w bits of all exponents in one multiplication. This technique is called the Simultaneous 2^w-ary method and the special case $w = 1$ is also known as the "Shamir's trick" [17]. We will denote this as the 2^w-ARY algorithm.

More formally, the algorithm consists of two phases:

1. *Precomputation* that can be done offline, in which we compute and store all the

$$\prod_{i=1}^{n} x_i^{E_i} \bmod N \tag{3}$$

for all $(E_1, \dots, E_n) \in \{0, \dots, 2^w - 1\}^k$.

2. *Evaluation*, a fast exponentiation-like algorithm that matches precomputation entries to batches w bits of each exponent and then squares w times in order to shift the computed exponent by w bits on the left. Algorithm 2 gives a pseudo-code of this part of the algorithm.

Algorithm 2: 2^w-ARY algorithm

```
Data: Bases $\{x_i\}_{i\in[1,n]}$, exponents $\{e_i\}_{i\in[1,n]}$,
      modulus $N$, precomputed values and window
      size $w$
Result: $A = \prod_{i=1}^{n} x_i^{e_i}$
1 $A \leftarrow 1$;
2 for $j \leftarrow \lfloor \frac{k-1}{w} \rfloor w$ to 0 by $w$ do
3     for $\_ \leftarrow 1$ to $w$ do
4         $A \leftarrow A^2 \bmod N$
5     if $(e_i[j+w-1, \dots, j] \neq 0, \forall i)$ then
6         $A \leftarrow \prod_{i=1}^{n} x_i^{e_i[j+w-1,\dots,j]} \bmod N$ {Multiply by
          the table entry};
7 return $A$
```

The precomputation phase has the following computational and storage complexity:

– Storing $2^{nw} - n - 1$ non-trivial elements of size λ.
– $2^{n(w-1)} - 1$ elements can be computed as a squaring of other entries.
– The remaining $2^{nw} - 2^{n(w-1)} - k$ elements require one modular multiplication each.

As for the evaluation phase, the algorithm requires

– $\lfloor \frac{k-1}{w} \rfloor w$ squarings.
– $k \cdot \frac{1 - \frac{1}{2^{nw}}}{w}$ modular multiplications.

3.3 Simultaneous sliding window algorithm (YLL)

Based on the previous simultaneous 2^w-ary method, Yen, Laih and Lenstra [26] have proposed an optimization which replaces the fixed window of the previous algorithm with a sliding window. It still consists in a precomputation phase, which can be done offline, followed by the exponentiation itself. We display the pseudocode of the multi-exponentiation phase in Algorithm 3. The optimization is twofold:

1. Before making the multiplication, the algorithm scans the most significant bits in the window that will be computed next to each exponent; while they are all zeros, the window is shifted to the left and we square the current exponentiation result (denoted by A in the pseudocode).
2. Once the window cannot be shifted anymore, we scan the least significant bits using a virtual cursor denoted J in the algorithm so that we multilply using only entries in which at least of exponent is odd. In that way, we save some precomputations by only computing entries with at least one odd element, which represents 25% of the precomputations.

Algorithm 3: YLL algorithm

```
Data: Bases $\{x_i\}_{i\in[1,n]}$, exponents $\{e_i\}_{i\in[1,n]}$,
      modulus $N$, precomputed values and window
      size $w$
Result: $A = \prod_{i=1}^{n} g_i^{e_i}$
1  $A \leftarrow 1$;
2  $j \leftarrow b - 1$;
3  while $j \geq 0$ do
4      if $\forall i \in \{1, \dots, n\}, e_i[j] = 0$ then
5          $A \leftarrow a^2; j \leftarrow j - 1$
6      else
7          $j_{new} \leftarrow \max(j - w, 1)$;
8          $J \leftarrow j_{new} + 1$
9          while $\forall i \in \{1, \dots, n\}, e_i[J] = 0$ do
10             $J \leftarrow J + 1$
11         for $i = 1$ to $n$ do
12             $E_i \leftarrow ei[j, \cdots, J]$
13         while $j \geq J$ do
14             $A \leftarrow A^2$;
15             $j \leftarrow j - 1$;
16             $A \leftarrow A \cdot \prod_{i=1}^{n} g_i^{E_i}$
17         while $j > j_n ew$ do
18             $A \leftarrow A^2$;
19             $j \leftarrow j - 1$;
20     return $A$
```

This algorithm requires:

– $2^{nw} - 2^{n(w-1)} - n$ non-trivial entries of size λ as for the precomputations.
– from $k - w$ up to $k - 1$ squarings.
– $k \cdot \frac{1}{w + \frac{1}{2^n - 1}}$ modular multiplications.

3.4 Basic interleaving method

Like the previous algorithms presented, the *basic interleaving method* [17] is based on square-and-multiply using a window of size w model but instead of precomputing all possible products of n bases with exponents of size w, it precomputes all $x_i^{E_i}$ where $E_i \in [0, 2^w - 1]$ and $i \in [1, n]$. During the evaluation, the returned value – denoted by A in the previous algorithms – is multiplied by a single base exponentiation $x_i^{E_i}$ instead of an aggregate of all of them. The consequence is that the evaluation takes n times more multiplications but

Algorithm	Precomputation	Evaluation
SEPARATE	N/a	k
2^w-ARY	$2^{nw} - 1 - n$	$\lfloor \frac{k-1}{w} \rfloor w + k\frac{1-\frac{1}{2^{nw}}}{w}$
YLL	$2^{nw} - 2^{n(w-1)} - n$	$k\left(\frac{1}{w+\frac{1}{2^n-1}} - 1\right) - 1$
Interleaving	$n2^{w-1}$	$nk\frac{1}{w+1}$

Table 1: Worst case number of multiplications expected for SEPARATE, 2^w-ARY, YLL and the interleaving algorithm.

the precomputations has a space and time complexity of $\mathcal{O}(2^w)$ as opposed to $\mathcal{O}(2^{nw})$ for 2^w-ARY and YLL. Theoretically, for an equivalent space used for precomputations, i.e, $w_{interleaving} = nw_{YLL}$, the interleaving algorithm performs slightly better, using at best around 10% less multiplications. However, this difference is decreasing as n gets larger and experiments resulting from our implementation showed that this small improvements are lost in practice due to complex bit scanning and filtering of large numbers. Furthermore, [17] confirms our conclusions showing that YLL is preferred in real implementations. For these reasons, we decided to focus our study on the previously mentioned algorithms.

3.5 Comparison between SEPARATE, 2^w-ARY and YLL

In the previous subsections, we provided formulas about the expected number of modular multiplications needed for each method. However, it can be hard to grasp how they compare to each other. For this reason, we show in Figure 1a the expected number of modular multiplications needed for SEPARATE, 2^w-ARY and YLL as a function of the exponent bit length. The plot is in logarithmic scale. The y-axis considers the sum of number of multiplications performed during the precomputation and evaluation phases. While the expected number of multiplications increases linearly for SEPARATE, the other techniques observe an initial overhead due to the precomputation phase. The asymptotic vertical offset between SEPARATE and windowed algorithms indicates that, for large values of the exponent, the difference is of a multiplicative constant.

Figure 1b shows the theoretical speedup of using 2^w-ARY and YLL compared to the naive optimal separate exponentiations (SEPARATE). We can identify the same behaviour as in the previous plot. The interest in this plot is rather the numerical data: we can observe that YLL saves asymptotically more than 2^w-ARY, achieving a speedup of around 5% bigger than the latter. This supports the interest of using YLL.

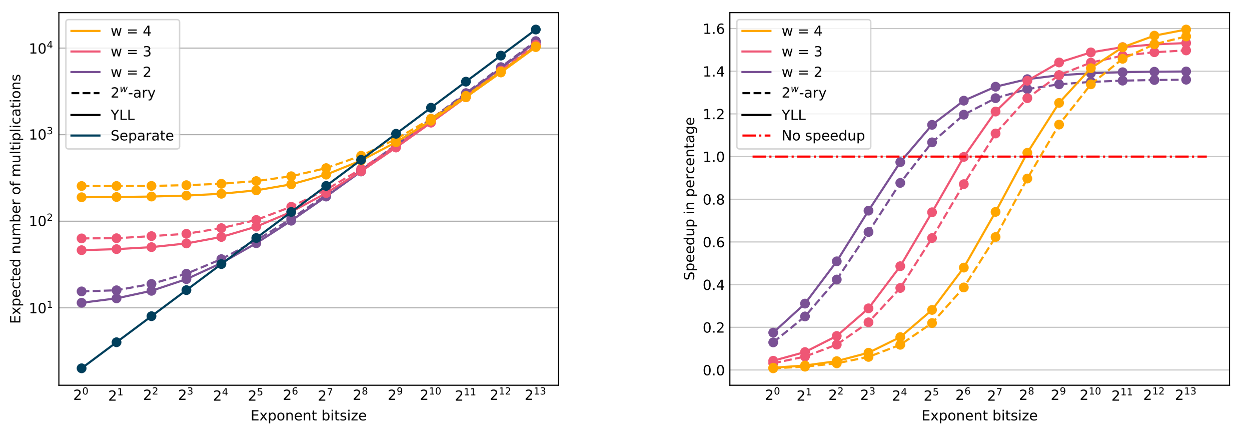
3.6 Additional remarks

It is important to highlight that modular arithmetic is not limited to only integer manipulations. Many of its operations, such as (ME) or (MME), can be generalized to arbitrary groups. This is important because the internal structure of specific groups may allow optimizations compared to the classical algorithms operating on integers. One of these optimizations is the possibility to easily invert an element in a group, i.e., for x in a group G, efficiently find x^{-1} such that $x \cdot x^{-1} = 1_G$ with 1_G being the neutral element of this group. Elliptic curves associated with the point addition operation is a relevant example of such groups [20]. This allows a whole new class of exponentiation and multi-exponentiation algorithms that use this feature by rewriting the exponents with negative values and then reducing the number of multiplications to perform [11]. However, efficient inversion is not available for all groups and this is particularly true for integers arithmetic. Then this motivates the study for algorithms that does not leverage this feature [26,17] because integer arithmetic is still widely used in modern cryptography, it being the foundation layer of the RSA cryptosystem [19] and multi-exponentiations are found in various applications such as verifiable delay functions [23,1] in which optimizing its verification time can be of great importance for time-sensitive uses.

More recently, after some early work by Chang and Lou [8,16,9], it has been shown that highly parallelizable modern processors allow for a new class of multi-exponentiation algorithms working on multiple cores[16,5]. In this paper we focus on single-core architectures and we leave multi-core ones as a future work: in fact, we firmly believe that there is still large interest into single-core applications as not all hardware can support parallelization, especially Internet-of-Things devices or embedded technologies that usually have very low computing power.

4 Montgomery multiplication

When it comes to modular arithmetic, the very straightforward way to do it is by applying the modulo operation as soon as the numbers handled grow in order not to spend time computing huge numbers that will anyway be reduced. For example computing $x = a \cdot b \bmod n$ can be done by computing $c = a \cdot b$ and then $x = c \bmod n$. However, doing so can be inefficient as the modulus operation on big numbers is a costly operation due to long divisions involved. In particular, hardware using RISC sets of instructions can spend much more time on the modulus operation than on the



(a) Expected number of multiplications in function of exponent length.

(b) Expected speedup comparison in function of exponent length.

Fig. 1: Theoretical comparison between SEPARATE, 2^w-ARY and YLL.

multiplication. Thus, some techniques have been designed to efficiently compute $a \cdot b \bmod N$ without involving the modulus operation itself. The most efficient ones are the Barrett reduction [2] and the Montgomery reduction [18]. As previously discussed, Montegomery reduction is better suitted for multi-exponentiation use, and is based on the idea to transform the numbers a and b into a special form in which the reduction can be efficiently computed.

First, an important mathematical result must be recalled. Let N be our modulus. If we take a natural number p such that $\gcd(p, N) = 1$, then the set $\{p \cdot x \bmod N\}_{\{x\in[1,N-1]\}}$ is a permutation of $[1, N-1]$. In other words, the modular multiplication by p is a bijective application and then can be inverted.

Then, if we assume that our modulus N is odd, which is often the case in cryptosystems such as RSA and has a size λ, then we can set $R = 2^\lambda$ and then $\gcd(N, R) = 1$. Then, for any x in G, we can define the *Montgomery representation of* x, $\bar{x} = x \cdot R \bmod N$ and the *Montgomery product* of $\bar{a}$ and $\bar{b}$ as $\bar{u} = \bar{a} \cdot \bar{b} \cdot R^{-1} \bmod N$ and it is easy to see that $\bar{u} = \overline{a \cdot b}$. Furthermore can also easily see that $\overline{a} + \overline{b} = \overline{a+b}$ then we can do arithmetic using the Montgomery representation of each number. In order to retrieve a number x from its Montgomery representation $\overline{x}$, we can just multiply it by R^{-1} and take the remainder modulo N.

The trick of using Montgomery representation is that when $R = 2^\lambda$, we have algorithms that can compute the Montgomery product *without* using the modulo operation, hence speeding the computation up. However, this technique is not interesting when the number of products is small amount due to the overhead required for the Montgomery conversion; conversely, it is very powerful when it comes to exponentiations in which a lot of multiplications are involved as the overhead is amortized by the lower computation cost in the Montgomery space.

5 Implementation results and discussion

In this section, we compare some of the algorithms presented in Section 3 to provide valuable insights depending on the values of sizes k and λ and on the hardware used. The rest of the Section is structured as follows: First, we introduce the experimental setup in Section 5.1, describing which algorithms are compared and what hardware is used. Then, we show in Section 5.2 the computing time of a specific (MME) problem. After that, in Section 5.3 we discuss about how using different hardware impact the algorithm performances. Finally, in Section 5.4 we perform sensitivity analysis on the bit sizes k and λ for each hardware and algorithm used.

5.1 Simulation setup

Our implementation uses the OpenSSL library[1] as it provides a convenient built-in interface for using Montgomery reduction. Conversely, the two major competing libraries, NTL and GMP, do not allow access to this interface. The algorithms implemented are:

– SEPARATE, which consists in merely computing $x^a \bmod N$, $y^b \bmod N$ separately using the built-in

[1] `https://www.openssl.org`

primitive `BN_mod_exp` for modular exponentiation and then multiplying them. It is important to note that the built-in exponentiation primitive uses Montgomery reduction under the hood.
- 2^w-ARY, using values of w in $\{2, 3, 4\}$.
- YLL, using values of w in $\{2, 3, 4\}$.

The (MME) problem considered in our simulations is of the form:

$$r \equiv x^a \cdot y^b (\text{mod} N), \tag{4}$$

where $x, y, N \in \mathbb{N}$ have size of λ bits and $a, b \in \mathbb{N}$ have a size of k bits. We invite the reader to note that we restrict the experimental analysis to the *double exponentiation problem*. Compared to Problem (MME), this is the case where $n = 2$. Verifiable delay functions [23, 1] and some other cryptographic schemes (e.g., the signature verification procedure in the digital signature standard, for example) need the case $n = 2$. When $n > 2$, we can use the same windowing techniques as the ones investigated in the paper or, we can segment the multi-exponentiation task into $n/2$ double-exponentiations and use directly the results presented here.

As anticipated, we evaluate the above multi-exponen-tiation techniques on various hardware:

- **Raspberry Pi:** We compared two generations of Raspberry Pi: the *Raspberry Pi 3 model B* (2016) and *Raspberry Pi 4 model B* (2020). The Raspberry Pi 3 represents the older but still largely used generation whereas the Raspberry Pi 4 is the newer version.
- **Intel Core:** It represents a late 2010s Intel CPU commonly found in general-purpose computers, using the x_86 instruction set and running in 64-bits. The exact CPU reference used is i7-7820HQ (8) @ 2.90GHz.
- **Apple M1:** This is the first version of the new ARM architecture released in 2020 that will equip all Apple computers in the future. We consider important to compare this hardware with the Intel CPU as this is the first ARM processor for general-purpose laptops.

A word on some technical details seems mandatory. In order to limit as much as possible the noise due to the multi-task nature of modern operating systems, we have tried to mitigate the side effects of preemption on our experiments. This necessary because of the very short timing measurements of multi-exponentiations, often in the order of a few dozen microseconds. Hence, getting our process preempted even for a few millisecond can pollute seriously our experiments. As we cannot kill every process running on the operating system, we decided to investigate processes priority.

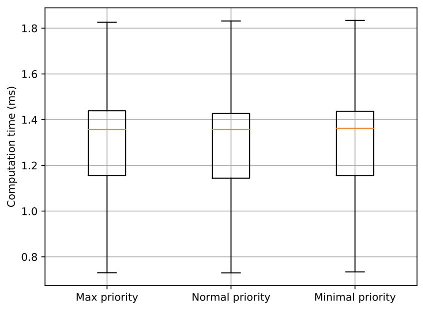


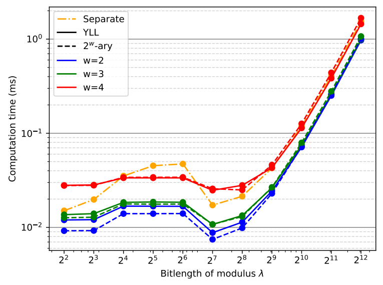
Fig. 2: Comparison of computation time of a batch of multi-exponentiations using three priority values: −20 (max priority), *default* set by the OS (normal priority) and 20 (min priority) as per UNIX priority definition.

Figure 2 shows on the y-axis the variation of time take to compute a batch of 100 multi-exponentiations with parameters $k = 512$, $\lambda = 1024$ using YLL with $w = 3$ on a M1 chip with three different levels of priority set when starting the processes using UNIX priorities, that is -20 (highest priority), default value set by the operating system (0) and 20 (lowest priority). We can see that the impact of priority during our experiments is negligible. We point out that the following precautions can help reduce the interferences: *i)* reducing the workload on the machine to limit as much as possible any other process running; *ii)* keeping CPU's temperature low to avoid thermal throttling. An efficient ventilation has proved to stabilize the measurements.

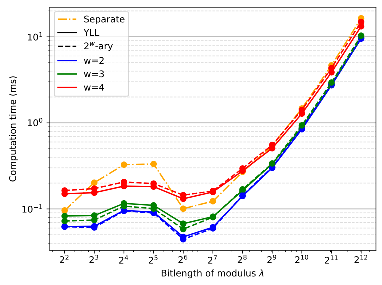
5.2 Computing time analysis

The first set of experiments shows the comparative behavior of multi-exponentiation algorithms with respect to the main parameters, such as exponent size k, modulus size λ and, to a lesser extent, the window size. In Figure 3 we display a set of five figures showing the time spent evaluating multi-exponentiations as in Eq. (4):

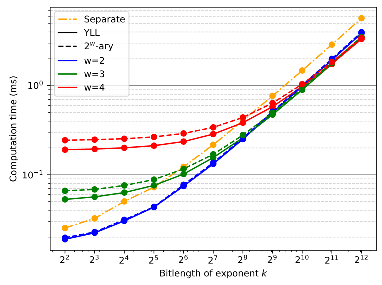
- Figure 3a represents the computation time using the Apple M1 as a function of the modulus bit-length λ, where the exponent size k is set to 256 bits.
- Figure 3b represents the computation time using the Raspberry Pi 4 as a function of the modulus bit-length λ, where the exponent size k is set to 256 bits.
- Figure 3c represents the computation time using the Apple M1 as a function of the exponent size k, where the modulus size λ is set to 2048.
- Figure 3d displays the speedup of using 2^w-ARY and YLL compared to SEPARATE, which is used as a



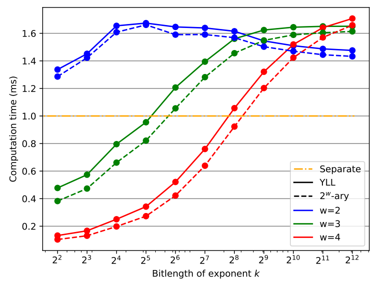
(a) Multiexponentiation algorithms for $k = 256$ on the Apple M1 chip with variation of parameters λ and w.



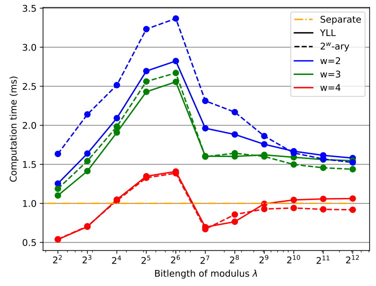
(b) Multiexponentiation algorithms for $k = 256$ on a Raspberry Pi 4 with variation of parameters λ and w.



(c) Multiexponentiation algorithms for $l = 2048$ on the Apple M1 chip with variation of parameters k and w.



(d) Speedup of multiexponentiation algorithms for $l = 2028$ on a Raspberry Pi 4 with variation of parameters λ and w.



(e) Speedup of multiexponentiation algorithms for $k = 256$ on a Raspberry Pi 4 with variation of parameters λ and w.

Fig. 3: Hardware comparison for YLL algorithm with $w = 3$.

baseline scenario, as a function of the exponent size k for $\lambda = 2048$.

– Figure 3e displays the speedup of using 2^w-ARY and YLL compared to SEPARATE, which is used as a baseline scenario, as a function of the modulus size λ for $k = 256$.

We decided to set $\lambda = 2048$ and $k = 256$ because we believe they represent reasonable values to be used in some real cryptosystems such as the Wesolowski's verifiable delay dunction construction[23]. Finally, in these figures, the 2^w-ARY (resp. YLL) method is represented by dashed (resp. solid) lines, which are blue in case $w = 2$, green if $w = 3$ and red if $w = 4$. Dash-dotted yellow lines represent the SEPARATE method.

The two first subplots reveal interesting behaviours. Considering Figure 3a, we can see that the variation over λ leads to a similar asymptotic behavior amongst all algorithms. It is important to note that, besides validating the theoretical findings, this plot shows that using these multi-exponentiation algorithms can be performed under the millisecond for realistic setup values like $\lambda = 2048$ and $k = 256$.

In Figure 3a we can see an increase from $\lambda = 2^2$ to $\lambda = 2^6$, suddenly followed by a decrease, and then the plot gradually increases again. This is due to the fact that $2^6 = 64$ is the size of a word on this 64-bit device and marks the threshold by which using a multi-precision library leads to improvements. This can be verified by looking at Figure 3b which displays the same parameters but for a Raspberry Pi 4 hardware, running on a 32-bits operating system. There we can see that the threshold is for $\lambda = 32$.

Figure 3c is very similar to Figure 1a, confirming the theoretical expectations. It confirms that using different values of w becomes interesting as k increases in practical settings, and implementation conditions do not sensibly affect the theoretical improvements.

Figure 3d is much less consistent with theory predicted in Figure 1b for low values of k, especially in the curve $w = 2$. However, for values of k larger than 2^8, results start to gain consistency. Furthermore, when comparing the asymptotic improvement factors, we can observe they are all larger than the predicted one by a factor of around 5%. The discrepancies seen for low values of k is due to libraries inner mechanisms that we do not have control over which get smoothed away with values of k over 2^8. Furthermore, we can see that YLL always outperforms 2^w-ARY by around 10%. Figure 3e is showing the expected behavior, i.e., that using different values of w for the same value of k does not lead to significant improvements for large values of λ. Someone implementing multi-exponentiations should then only consider k when picking a value of w. However, the algorithm used, i.e., 2^w-ARY or YLL makes a difference, with YLL performing better for large values of λ.

5.3 Hardware Comparison

An aspect that we considered important in our analysis was the different behavior depending on the hardware architectures. This can help anyone implementing multi-exponentiations to understand what are the expected discrepancies in case of heterogeneous ecosystems. This is displayed in Figure 4. This analysis is similar to the one performed in the previous subsection: while in Figure 3 we study how fast multi-exponentiation algorithms perform with a variation on the parameters k and λ, here we restrict our study by setting $w = 3$ and focusing on the impact of different hardware. We have already started comparing hardware by showing a behavior happening when using 32-bits or 64-bits operating systems but in this section we are going to present a more exhaustive study of hardware influence.

Figure 4a shows the computation time of YLL algorithm with $w = 3$ as a function of the exponent size k with $\lambda = 2048$. Each line represents a different hardware as depicted in the legend while Figure 4b shows the same data in the form of speedup compared to the SEPARATE algorithm, used as a baseline. Figures 4c and 4d display the same information but performing a sensitivity analysis over the modulus size λ where k is set equal to 256.

The high-level behavior of the plots follows the observations made in the previous subsection. We can see that using different hardware can only be observed via vertical shifts in performances as clearly displayed by Figure 4a. Moreover, in Figures 4c and 4d we can observe an horizontal offset between the 32 bits and 64 bits devices as devised in Section 5.2.

Another interesting observation is that in Figure 4b, the two Raspberry Pi devices have better speedup for YLL than the Intel Core and Apple M1 processors. The same behavior can be observed in Figure 4d in an asymptotic way although there is a rather chaotic behavior for low values of λ. This is a motivation for further research to understand how different hardware perform multi-exponentiation algorithms.

In Table 2 we summarize the results of our experiments with respect to the different hardware for SEPARATE, 2^w-ARY and YLL with values of w in $\{2, 3, 4\}$ when $\lambda = 2048$ and $k = 256$. The last column represents the speedup achieved by the fastest algorithm compared to the SEPARATE for each line.

We can see the same global behavior as in Section 5.2 but taking a closer look at the actual numeric values is instructive. Each cell contains the minimum of around a hundred multi-exponentiation runs for each configuration and algorithm. A first observation tells us that, without surprise, the desktop-grade processors perform much better than the Raspberry Pi devices, outperforming them by a factor of around $\times 11$ for the Raspberry Pi 4 and $\times 19$ for Raspberry Pi 3, and these ratios are consistent throughout the whole data. Although this result should be of no surprise for anyone, we considered having precise measurements for this specific application would be of interest to anyone building applications using heterogeneous hardware.

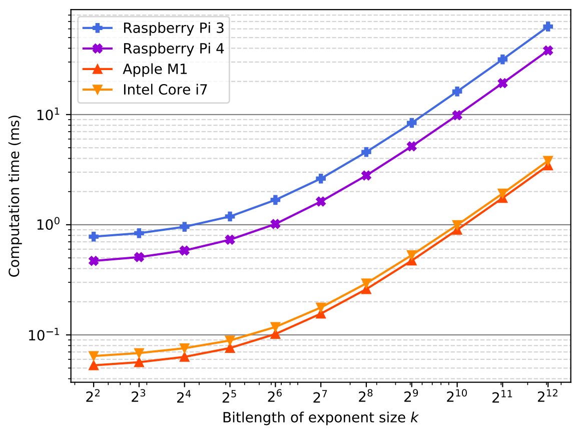
A second observation shows that the behavior is consistent for all devices when using different algorithms, as showcased in Figure 4c. We finally can see that the maximum expected speedup is 1.75 for this specific application.

5.4 Best multi-exponentiation algorithm

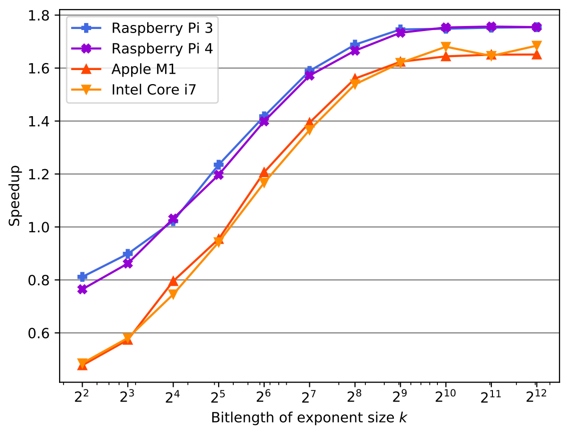
A very instructive way of representing data is to show which algorithm performs the best depending on the parameters k and λ. Hence, in this subsection we plot the fastest algorithm for each value of k and λ in $\{2^i | i \in [2..13]\}$ depending on the hardware. We present the results as a grid for which each cell contains a color corresponding to the fastest algorithm. The different hues indicate a change of window size with $w = 2$ in green, $w = 3$ in blue and $w = 4$ in purple while the naive implementation is in red. The shade in turn will show a change of algorithm as described in the plot's legend.

Figures 5a, 5b, 5c, 5d show the best algorithm for respectively the Mac Intel, Mac ARM, Raspberry Pi 3 and Raspberry Pi 4. One can observe two patterns, each one taking place along one of the two axes and thus orthogonal with each other. When the exponent size k varies, adapting the value of the window w is important. Conversely, depending on the value of the modulus size λ, a change of algorithm from 2^w-ARY to YLL leads to better results, and we notice that this pattern is consistent for any hardware tested.

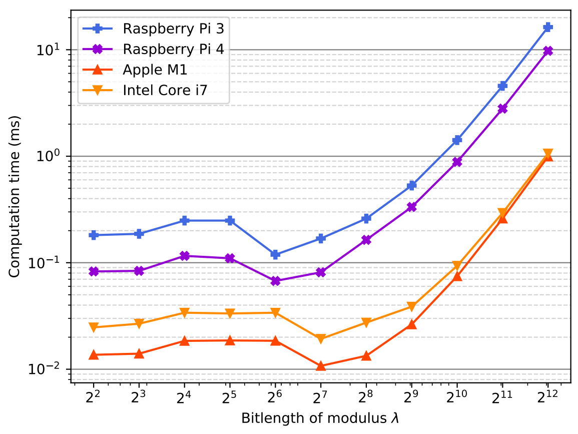
The variation along the k axis is explained by the fact that when performing multi-exponentiation algorithms, there is a certain overhead in precomputations which is linear with 2^w, hence it only depends on the value w, while the evaluation part is a function of growing with k and decreasing with w. We can then see that



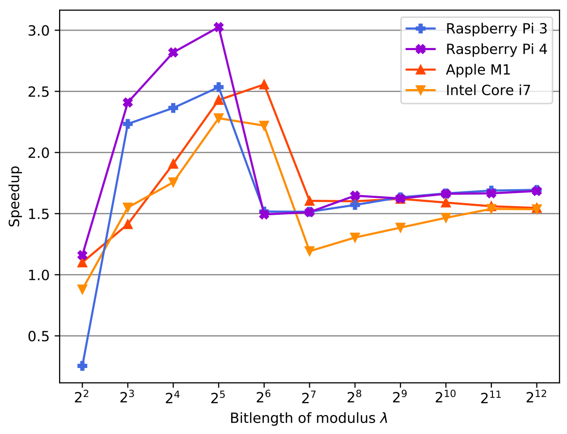
(a) Computation time when varying k with $l = 2048$.



(b) Computation time when varying k with $l = 2048$.



(c) Computation time when varying λ with $k = 256$.



(d) Speedup when varying λ with $k = 256$.

Fig. 4: Hardware comparison for YLL algorithm with $w = 3$.

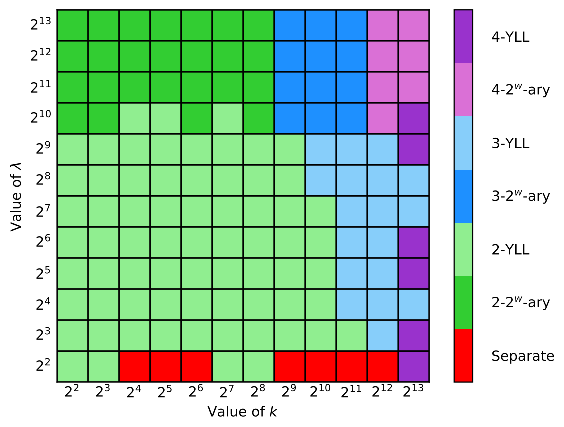
Hardware	SEPARATE	2^w-ARY			YLL			Max Speedup
		w=2	w=3	w=4	w=2	w=3	w=4	
Mac ARM	0.41	0.26	0.28	0.38	0.25	0.26	0.38	×1.58
Intel i7	0.45	0.29	0.31	0.51	0.28	0.29	0.45	×1.61
Raspberry Pi 3	7.72	4.56	4.83	7.06	4.41	4.57	6.30	×1.75
Raspberry Pi 4	4.66	2.81	2.95	4.34	2.74	2.80	3.87	×1.66

Table 2: Multi-exponentiation values for $\lambda = 2048$ and $k = 256$ on Mac ARM.

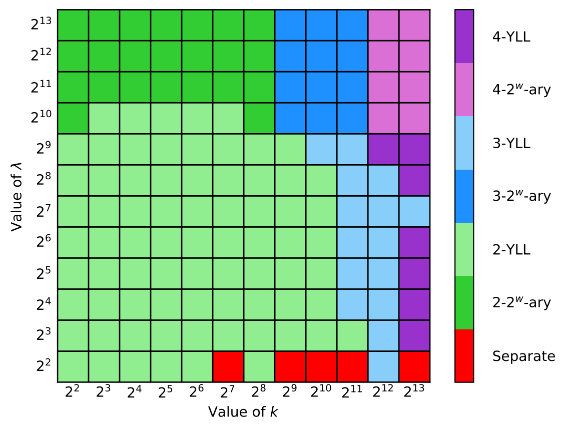
increasing k diminishes the relative overhead of the precomputation phase and at some point it becomes interesting to increase the value of w in order to leverage the speedup that it brings in the actual evaluation part. That explains the pattern shift on the k axis.

Concerning the λ axis, the rationale is that increasing the modulus length only impacts the computation time of modular multiplications in the same way for all algorithms and for all values of w. Then it becomes more efficient to optimize the parts of the algorithm that are not related to the actual modular multiplications such as filtering and bit testing. In that sense, the 2^w-ARY method has a simpler mechanic than the sliding window one. Thus, it is better to use for smaller values of λ. However for big values, we can benefit from the reduced number of modular multiplications performed during the sliding window method.

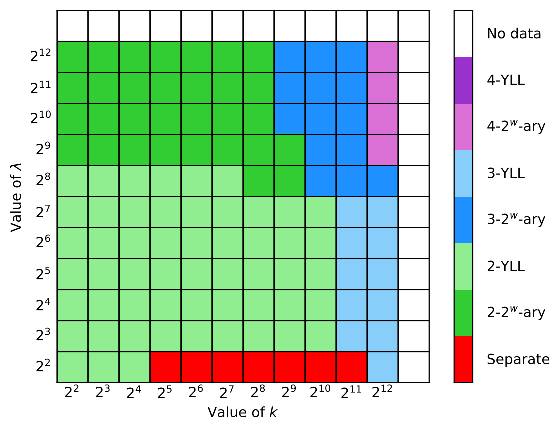
Another interesting point is that the pattern shifts around the same value of the exponent k for any hardware. However, we can see that for the λ-axis, the



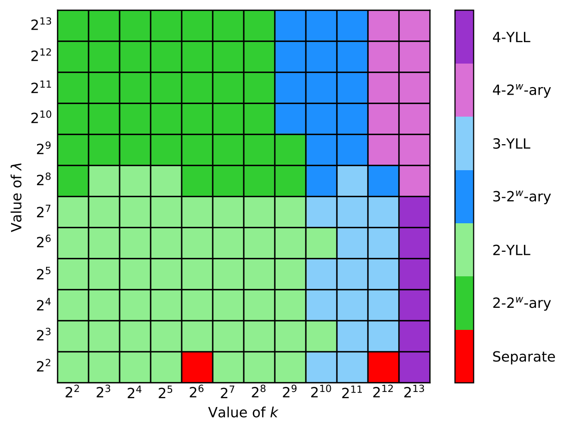
(a) Best algorithm as a function of λ and k for Intel Core.



(b) Best algorithm as a function of λ and k for Apple M1.



(c) Best algorithm as a function of λ and k for Raspberry Pi 3.



(d) Best algorithm as a function of λ and k for Raspberry Pi 4.

Fig. 5: Heat map showing the fastest multi-exponentiation algorithm per hardware depending on modulus and exponent sizes.

Mac Intel and ARM have a switching value around 2^{10} whereas the Raspberry Pi 3 and 4 have a switching value around 2^8. The rationale is that the default Raspberry Pi operating system works in 32-bits mode even though he Raspberry Pi 3 and 4 CPU have 64-bits capabilities. This is due to backward compatibility issues. This makes the difference in computation time bigger for big numbers modular multiplication than all other computations involved because the word length is smaller. As mentioned above, difference between modular multiplication and the other operations performed in important only on the λ-axis, thus the behavior seen on these plots.

6 Conclusion

One of the main points of our paper is to demonstrate that there is a need to rethink the use of performance metrics in considering the (MME) problem (and, of course, many other essential cryptographic primitives). The algorithms proposed in the past have been designed with one chief goal only – minimize as much as possible the number of modular multiplications. But a small reduction of the number of modular multiplications can be easily overshadowed by other factors like more complex architectures, larger number of precomputations, less opportunity to use parallelization, to name a few. Our research unveils the complicated relationship between many of these factors and presents the best possible algorithms for a large number of cases essential in cryptographic applications. We aim our efforts specifically on the applications related to verifiable delay functions, but other "users" of multi-exponentiation algorithms can directly benefit from the findings presented here.

We have presented an implementation study of three algorithms to solve double exponentiation, SEPARATE, 2^w-ARY, and YLL. We have focused our study on understanding how computation time is affected by problems parameters, k and λ, respectively being the exponents bit-size and the modulus bit-size and understanding which optimal parameter w, that is the window size in 2^w-ARY and YLL, to pick for a given pair (k, λ). We have provided extensive experimental results for these three algorithms, considering four different hardware to understand how platform-specific behavior happen.

Although experimental results follow theoretical predictions, there are implementation side effects that are highlighted in this paper. First, running a 32-bits or 64-bits operating system does impact the performances and change how algorithms compare to each other. Second, although YLL is always theoretically faster than 2^w-ARY, performing exponents scanning and bit filtering induce an overhead which is not taken into account in predictions. Thus, for low values of k, the 2^w-ARY algorithm actually performs better than YLL.

We show that the two parameters k and λ have an orthogonal role in double exponentiation performances. An increase λ yields longest modular multiplications whereas a larger value of k induces a more important number of modular multiplications performed and make bigger precomputations affordable, hence increasing the value of optimal w.

We finally show that results are pretty consistent regardless the considered hardware, although using a 32-bits or 64-bits operating system changes the threshold value of λ for which YLL performs better than 2^w-ARY. This makes algorithm selection easy for real applications.

This paper opens some perspectives for future work. Important topics yet to understand include understanding how generalizing to $n > 2$ exponentiations behaves, consider the case in which precomputations can be done offline and then only consider only the evaluation phase or have a study focusing on a very large variety of hardware to have an even better understanding of platform-dependent effects. Another topic would be to study side channel resistant algorithms. Binary GCD [25] is a good example of such algorithm.

6.1 Generalizations and additional considerations

Over the last few years several new cryptographic schemes that make use of different algebraic structures, like quaternions, have been proposed. The task of computing modular exponentiation and multi-exponentiations for quaternions has been considered in [22] and [24]. A similar task for computing multi-exponentiations in the cases of matrices (it is essential in some automatic control problems [3]) was also posed by some researchers. What is the main difference in these cases? It is the implicit use of commutativity in all of the already existing algorithms! Indeed, the computation of $\mathbf{W} = \mathbf{A}^2\mathbf{B}^2$ ($\mathbf{A}$, $\mathbf{B}$ square matrices) has to be done as $\mathbf{W} = (\mathbf{AA})(\mathbf{BB})$; the non-commutative nature of the matrix multiplication prevents us to use tricks like computing the same statement as $(\mathbf{AB})^2 = ((\mathbf{AB})(\mathbf{AB}))$. For readers who might be interested in exploring these fascinating problems, we will provide here two basic definitions: a matrix $\mathbf{A}$ is called nilpotent if there exist a positive integer, n, such that $\mathbf{A}^n = \mathbf{0}$; a pair of matrices $(\mathbf{A}, \mathbf{B})$ is called mortal if there exist a pair of positive integers (m, n) such that $\mathbf{A}^m\mathbf{B}^n = \mathbf{0}$. Whilst testing if a matrix $\mathbf{A}$ is nilpotent is relatively easy, testing mortality of pair of matrices is extremely difficult – there are no provable bounds on the exponents (m, n) and matrix double-exponentiation computation cannot use many of the computational tricks presented in our paper, due to the non-commutativity. So, for these cases one has to use a radically different approach. In cryptographic settings this means that one has to use either separate exponentiations (that would be unfortunate and a serious drawback to the performance of quaternion-based cryptographic schemes like the one in [22], for example) or completely different multi-exponentiations techniques have to be found. We hope this remark of ours will spark the attention of the cryptographic algorithms designers to take a deeper look into this under-researched task.

Acknowledgements We would like to thank some people that helped elaborating this paper. First, we would like to thank Pr. Paul Zimmermann from Loria, France, who took a lot of time to help us understand how multiprecision libraries work. Then Colin Walter who clarified a lot of details on Montgomery reduction implementation. Thank you also to Wolgang Welz from the IOTA Foundation for his engineering advices that helped us define the scope of this paper. Finally, thank you to Ilan Zajtmann for lending us some computation time on his Apple M1 computer to help us include cutting edge technology in this survey.

References

1. Attias, V., Vigneri, L., Dimitrov, V.: Implementation Study of Two Verifiable Delay Functions. In: Tokenomics, 6, pp. 1–6 (2020)
2. Barrett, P.: Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 263 LNCS (1987). DOI 10.1007/3-540-47721-7_24
3. Blondel, V.D., Tsitsiklis, J.N.: When is a pair of matrices mortal? Information Processing Letters **63**(5) (1997). DOI 10.1016/s0020-0190(97)00123-3
4. Blum, T., Paar, C.: Montgomery modular exponentiation on reconfigurable hardware. Proceedings - Symposium on Computer Arithmetic (1999). DOI 10.1109/arith.1999.762831
5. Borges, F., Lara, P., Portugal, R.: Parallel algorithms for modular multi-exponentiation. Applied Mathematics and Computation **292**, 406–416 (2017). DOI 10.1016/j.amc.2016.07.036
6. Bosselaers, A., Govaerts, R., Vandewalle, J.: Comparison of three modular reduction functions. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 773 LNCS (1994). DOI 10.1007/3-540-48329-2_16
7. Brent, R., Zimmermann, P.: Modern Computer Arithmetic. October. Cambridge University Press, Cambridge (2010). DOI 10.1017/cbo9780511921698
8. Chang, C.C., Lou, D.C.: Parallel computation of the multi-exponentiation for cryptosystems. International Journal of Computer Mathematics **63**(1-2) (1997). DOI 10.1080/00207169708804548
9. Chang, C.C., Lou, D.C.: Fast Parallel Computation of Multi-Exponentiation for Public Key Cryptosystems. In: Parallel and Distributed Computing, Applications and Technologies, PDCAT Proceedings (2003). DOI 10.1109/pdcat.2003.1236459
10. Dimitrov, V.S., Jullien, G.A., Miller, W.C.: An algorithm for modular exponentiation. Information Processing Letters **66**(3), 155–159 (1998). DOI 10.1016/s0020-0190(98)00044-1. URL `https://www.sciencedirect.com/science/article/abs/pii/S0020019098000441`
11. Dimitrov, V.S., Jullien, G.A., Miller, W.C.: Complexity and fast algorithms for multiexponentiations. IEEE Transactions on Computers **49**(2), 141–147 (2000). DOI 10.1109/12.833110. URL `http://ieeexplore.ieee.org/document/833110/`
12. Euclid: The Elements of Euclid, 2nd edn. Dover Publications Inc., New York, New York, USA (1956)
13. Gueron, S.: Efficient software implementations of modular exponentiation. Journal of Cryptographic Engineering **2**(1) (2012). DOI 10.1007/s13389-012-0031-5
14. Koç, Ç.K.: Montgomery Arithmetic, pp. 799–803. Springer US, Boston, MA (2011). DOI 10.1007/978-1-4419-5906-5_38. URL `https://doi.org/10.1007/978-1-4419-5906-5_38`
15. Kochergin, V.V.: On Bellman's and Knuth's Problems and their Generalizations. Journal of Mathematical Sciences (United States) **233**(1) (2018). DOI 10.1007/s10958-018-3928-4
16. Lou, D.C., Chang, C.C.: An efficient divide-and-conquer technique for parallel computation of modular multi-exponentiation. Computer Systems Science and Engineering **15**(2) (2000)
17. Möller, B.: Algorithms for multi-exponentiation. In: Vaudenay, Serge, Youssef, A. M. (eds.) Selected Areas in Cryptography, vol. 2259, pp. 165–180. Springer Berlin Heidelberg (2001). DOI 10.1007/3-540-45537-x_13. URL `https://www.bmoeller.de/pdf/multiexp-sac2001.pdf`
18. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation **44**(170), 519–521 (1985)
19. Rivest, R.L., Shamir, A., Adleman, L.: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM **21**(2), 120–126 (1978). DOI 10.1145/359340.359342
20. Roberto M., A.R.U.o.B.: The Complexity of Certain Multi-Exponentiation Techniques in Cryptography. Journal of Cryptology pp. 357–373 (2005)
21. Van Meter, R., Itoh, K.M.: Fast quantum modular exponentiation. Physical Review A - Atomic, Molecular, and Optical Physics **71**(5) (2005). DOI 10.1103/PhysRevA.71.052320
22. Wang, B., Hu, Y.: Signature scheme using the root extraction problem on quaternions. Journal of Applied Mathematics **2014** (2014). DOI 10.1155/2014/819182
23. Wesolowski, B.: Efficient verifiable delay functions. In: Ishai, Yuval, Rijmen, Vincent (eds.) Advances in Cryptology – EUROCRYPT 2019, vol. 11478 LNCS, pp. 379–407. Springer International Publishing (2019). DOI 10.1007/978-3-030-17659-4_13
24. Yagisawa, M.: A Digital Signature Using Multivariate Functions on Quaternion Ring. IACR Cryptology ePrint Archive **2010**(2), 352 (2010). URL `http://dblp.uni-trier.de/db/journals/iacr/iacr2010.html#Yagisawa10`
25. Yen, S.M., Chen, C.N., Moon, S.J.: Multi-exponentiation algorithm based on binary GCD computation and its application to side-channel countermeasure. Journal of Cryptographic Engineering **2**(2), 99–110 (2012). DOI 10.1007/s13389-012-0032-4
26. Yen, S.M., Laih, C.S., Lenstra, A.K.: Multi-exponentiation. IEE Proceedings: Computers and Digital Techniques **141**(6), 325–326 (1994). DOI 10.1049/ip-cdt:19941271. URL `https://digital-library.theiet.org/content/journals/10.1049/ip-cdt{_}19941271`