Efficient Modular Multiplication*

Joppe W. Bos[1], Thorsten Kleinjung[2] and Dan Page[3]

[1] NXP Semiconductors, Leuven, Belgium
[2] EPFL, Lausanne, Switzerland
[3] University of Bristol, Bristol, United Kingdom

This chapter is concerned with one of the fundamental building blocks used in modern public-key cryptography: modular multiplication. Speed-ups applied to the modular multiplication algorithm or implementation directly translate in a faster modular exponentiation for RSA or a faster realization of the group law when using elliptic curve cryptography.

This chapter outlines the most commonly used modular multiplication method *Montgomery multiplication* for generic moduli as well as different techniques when "special" moduli of a particular shape are used. Moreover, we study approaches which might produce errors with a very small probability. Such faster "sloppy reduction" techniques are especially beneficial in cryptanalytic settings. We look at this from both a historical as well as an applied implementation perspective. The best approach to implement modular multiplication on a modern 64-bit architecture with advanced single-instruction, multiple data instruction set extensions is, for example, quite different from the best approach on resource constrained embedded devices.

Throughout this chapter we focus on the cryptographic setting unless we specifically discuss an algorithm for cryptanalysis. Contrary to many mathematical software applications, the running time of a cryptographic implementation (and hereby also the modular multiplication) should avoid secret-data-dependent branches and secretly indexed memory access. Such *constant time* implementations are one of the basic countermeasures against timing attacks: advanced techniques which use information about the running time of the target algorithm to extract the used private key. Such attacks are part of a larger family of attacks known as side-channel attacks.

Throughout this chapter we represent a wn-bit non-negative integer X in the so-called *radix*-2^w *representation*,

$$X = \sum_{i=0}^{n-1} x_i 2^{wi} \tag{1}$$

where $0 \leq x_i < 2^w$. We denote x_i the i-th word of the integer X.

1 Montgomery multiplication

In order to accelerate the modular multiplication on modern computer platforms Peter Montgomery introduced a modular reduction technique now known as *Montgomery reduction* [43]. The main idea behind this approach is to change the representatives of the residue classes and change the modular multiplication accordingly.

Let N be an odd wn-bit integer, here we assume w is the *word size* of the target computer platform (say 32- or 64-bit for modern architectures) and the modulus N can be represented by an array of n such computer words. More precisely, instead of computing the modular multiplication $A \cdot B \bmod N$ the *Montgomery multiplication* computes $\mathrm{MontMul}(A, B) = A \cdot B \cdot 2^{-wn} \bmod N$. In order to use this modular multiplication method one needs to change the representation of the inputs.

*This material will be published in revised form in Computational Cryptography edited by Joppe W. Bos and Martijn Stam and published by Cambridge University Press. See www.cambridge.org/9781108795937.

Algorithm 1 The radix-2^w interleaved Montgomery multiplication algorithm. Compute $A \cdot B \cdot 2^{-wn}$ modulo the odd modulus N using the pre-computed Montgomery constant $\mu = -N^{-1} \bmod 2^w$.

Input: $A = \sum_{i=0}^{n-1} a_i 2^{wi}, B, N$ such that N is odd,
$0 \le a_i < 2^w, \quad 0 \le A, B < 2^{wn}, \quad 2^{w(n-1)} \le N < 2^{wn}$.
Output: $C \equiv A \cdot B \cdot 2^{-wn} \bmod N$ such that $0 \le C < N$.
1: $C \leftarrow 0$
2: **for** $i = 0$ to $n - 1$ **do**
3: $\quad C \leftarrow C + a_i B$
4: $\quad q \leftarrow \mu C \bmod 2^w$
5: $\quad C \leftarrow (C + Nq)/2^w$
6: **end for**
7: **if** $C \ge N$ **then**
8: $\quad C \leftarrow C - N$
9: **end if**
10: **return** C

Given a wn-bit modulus N, we define the *Montgomery form* of an integer A to be $\tilde{A} = A \cdot 2^{wn} \bmod N$. This change of residue class ensures that the multiplication of two inputs in Montgomery form corresponds to the desired result in Montgomery form since

$$\mathrm{MontMul}(\tilde{A}, \tilde{B}) \equiv \tilde{A} \cdot \tilde{B} \cdot 2^{-wn} \equiv A \cdot 2^{wn} \cdot B \cdot 2^{wn} \cdot 2^{-wn} \equiv \widetilde{A \cdot B} \pmod{N}.$$

This change of representation is performed since computing the Montgomery multiplication can be done efficiently on modern computer architectures where multiplications and exact divisions by powers of two correspond to shifting the number to the left or right, respectively.

Montgomery multiplication uses the pre-computed value $\mu = -N^{-1} \bmod 2^{wn}$. Then, if we compute the Montgomery reduction of an integer C such that $0 \le C < N^2$ the idea is to add the multiple $N \cdot (\mu \cdot C \bmod 2^{wn})$ to C. Adding a multiple of N does not change the outcome modulo N and the computing a reduction modulo 2^{wn} is for free on modern computer architectures: just take the wn least significant bits. Adding this multiple of N ensures

$$\begin{aligned} C + N \cdot (\mu \cdot C \bmod 2^{wn}) &\equiv C - N \cdot \left(N^{-1} \cdot C \bmod 2^{wn}\right) \\ &\equiv C - C \equiv 0 \pmod{2^{wn}} \end{aligned}$$

Hence, $C + N \cdot (\mu \cdot C \bmod 2^{wn})$ is divisible by 2^{wn} which can be computed by shifting wn positions to the right avoiding an expensive division operation. Moreover, after this division by 2^{wn} the result has been reduced to at most $2N$ since

$$0 \le \frac{C + N \cdot (\mu \cdot C \bmod 2^{wn})}{2^{wn}} < \frac{N^2 + N \cdot 2^{wn}}{2^{wn}} < 2 \cdot N$$

because $N < 2^{wn}$. This means a completely reduced result in $[0, N]$ can be computed with an additional conditional subtraction.

In many practical implementations of Montgomery multiplication the *interleaved* Montgomery multiplication algorithm is used. This approach merges the multiplication and reduction: after multiplying one computer word of A with the entire input of B the result is reduced modulo N. This has the advantage that the maximum size of the intermediate result remains significantly smaller: $n + 2$ computer words instead of the $2n + 1$ required when computing the full product $A \cdot B$ first and doing the Montgomery reduction next. This approach is outlined in Algorithm 1.

2 Arithmetic for RSA

Although embellishments such as padding (e.g., via OAEP [4], per PKCS #1 [31]) are important from a security perspective, textbook RSA [52] can be described in terms of arithmetic in the

multiplicative group $(\mathbb{Z}/N\mathbb{Z})^*$ as follows:

1. generation of a public and private key pair: given a security parameter λ

$$\textsc{KeyGen}(\lambda) \;=\; \begin{cases} \text{select random } \frac{\lambda}{2}\text{-bit primes } p \text{ and } q \\ \text{compute } N = p \cdot q \\ \text{compute } \Phi(N) = (p-1) \cdot (q-1) \\ \text{select random } e \in (\mathbb{Z}/N\mathbb{Z})^* \text{ such that } \gcd(e, \Phi(N)) = 1 \\ \text{compute } d = e^{-1} \pmod{\Phi(N)} \\ \text{return public key } (N, e) \text{ and private key } (N, d) \end{cases}$$

2. encryption of a plaintext $m \in (\mathbb{Z}/N\mathbb{Z})^*$:

$$\textsc{Enc}((N, e), m) \;=\; m^e \pmod{N}$$

3. decryption of a ciphertext $c \in (\mathbb{Z}/N\mathbb{Z})^*$:

$$\textsc{Dec}((N, d), c) \;=\; c^d \pmod{N}$$

The description implies that efficiency of the underlying arithmetic and RSA itself are directly related. This fact could be viewed as an advantage, because improvement of the former will clearly yield improvement of the latter, *or* as a challenge: *if* the former cannot be efficient enough, RSA becomes impractical. Rivest, Shamir, and Adleman *themselves* seem to have been well aware of this challenge. For example, preceding modern advice with respect to the *co*-design of cryptographic constructions and their implementation (see, e.g., the RWC'15 invited talk by Bernstein [6, Page 24]), their research paper [52] included overt focus on the latter. [52, Section IV.A], for example, discusses efficient realisation of encryption and decryption operations via an explicit left-to-right binary or "square-and-multiply" algorithm (see, e.g., Knuth [35, Section 4.6.3] or Gordon [26, Section 2.1]) for modular exponentiation. As one of the first published (cf. classified work by Cocks [54, Chapter 6]) public-key encryption schemes, it seems likely that one rationale for them to do so would be to rebut any claims of impracticality with respect to the technology landscape of that era. In fact, it *remains* challenging to implement efficient modular arithmetic and hence RSA on *both* higher-end platforms (e.g., due to increasingly demanding workloads), *and* lower-end platforms (which are, e.g., constrained with respect to computation and storage).

A vast range of literature has been dedicated to addressing such challenges more generally, so it seems reasonable to claim that RSA has acted as a driver for innovation with respect to modular arithmetic. Such innovation spans both software and hardware, of course, but, as noted as an aside in [52, Section IV.A], special-purpose hardware was (and still is) an attractive way to deliver efficiency. Rivest [50] describes an ASIC-based implementation by himself, Shamir, and Adleman; as the first such implementation of RSA, it provides valuable insight into the state-of-the-art in efficient modular arithmetic of that era. The implementation is best described as a co-processor for multi-precision integer arithmetic, requiring 40000 transistors within a single 42 mm^2 chip. The co-processor could be directed to compute various operations on operands stored in eight 512-bit registers. A limited set of operations, e.g., multiply-accumulate, were supported directly by a 512-bit ALU, and others via a 224-word micro-code program. Use of a micro-coded approach allowed more complex, number theoretic operations and, crucially, modular multiplication and exponentiation as required by RSA. Pre-dating techniques such as Montgomery multiplication (see Section 1 on page 1), modular reduction would likely have been realised using integer division, which, in turn, likely used a shift-and-subtract approach [48, Section 13.1] supported by the ALU. Operating at 4 MHz, the co-processor was able to "perform RSA encryption at rates in excess of 1200 bits/second". Although the manufactured result never worked reliably, the design process is also remarkable in that it leveraged a purpose-built, LISP-style HDL, *and* motivated subsequent work by Rivest on the theory and practice of place-and-route [53].

Many facets of the technology landscape have changed since publication of RSA in 1978, not least the increased societal awareness, and importance of cryptography in general. Ultimately,

however, RSA has remained largely (i.e., with caveats per [11]) secure and so has been widely commercialised (see, e.g., [51]) and deployed during what is a rich, 40+ year history. This demonstrates that associated challenges with respect to efficient modular arithmetic have at least been mitigated if not solved, with any resulting innovations refined and capitalised on by more recent constructions. In this Section we survey a very selective, limited subset of that history, focusing largely, but not exclusively on RSA-specific approaches.

2.1 Capitalising on special-form moduli

The selection of special-form parameters is a common, general optimisation strategy; doing so allows specialisation of associated algorithms, e.g., eliminating any overhead required to cope with the general case. Indeed, certain RSA parameters can be optimised in this way. The use of a short and low Hamming weight encryption exponent, e.g., $e = 65537$, is common: this replaces a general-purpose modular exponentiation with a short, fixed sequence of modular multiplications, for example. Equally, it is plausible to focus on the modulus N as a strategy for optimising the modular multiplications themselves. Where Montgomery multiplication is used per Section 1, imposing a special-form on the pre-computed value $\mu = -N^{-1} \bmod 2^w$ represents one such strategy. More specifically, a multiplication by μ (e.g., $\mu \cdot C$ in line 4 of Algorithm 1) will be more efficient, even trivial, if we can select N such that $\mu = \pm 1$. Crucially, however, the impact of *over*-optimisation on the security properties of RSA must be carefully considered. Selecting *too* small an e, e.g., $e = 3$, should be avoided due to the attack of Wiener [62], for example. Likewise, any special structure in N could, intuitively, be exploited during an attempt to factor it.

2.1.1 Selecting a special-form modulus

Consider three classes of special-form N, namely

$$N = p \cdot q = \begin{cases} f_{MSB} \parallel r & \Rightarrow \text{ 1) pre-determined MSBs} \\ r \parallel f_{LSB} & \Rightarrow \text{ 2) pre-determined LSBs} \\ f_{MSB} \parallel r \parallel f_{LSB} & \Rightarrow \text{ 3) pre-determined MSBs and LSBs} \end{cases}$$

where f_{MSB} and f_{LSB} denote *fixed* (or constrained) values and r denotes a *random* (or unconstrained) value, all of suitable length: given a choice of f_{MSB} and/or f_{LSB}, these classes imply the MSBs and/or LSBs of N are pre-determined (i.e., selected) during key generation. The question is, how?

Vanstone and Zuccherato [58] describe an approach for the first and second classes, i.e., selection of some t MSBs ([58, Section 2]) or LSBs ([58, Section 7]). Citing a range of prior art, Lenstra [37] reviews an "obvious and straight-forward trick" that affords both generalisation *and* improved efficiency by a) supporting all three classes, and b) avoiding the need for factorisation as a sub-step. Although [37, Section 4] provides a high-level description of how the three classes of special-form N could be capitalised on in the context of RSA, selection of LSBs, in particular, directly enables the strategy outlined above: one simple selects the w LSBs such that $-1 \equiv N \pmod{2^w}$, for example, which then naturally implies $\mu = 1 \equiv -N^{-1} \pmod{2^w}$ as required.

Both Vanstone and Zuccherato [58, Section 11] and Lenstra [37, Section 4] consider the security implications of selecting an associated N, framed within the context of contemporary factoring algorithms. A (very) informal summary would be that such special-form N have no negative impact on the security of RSA, *provided* that the fixed portions are not *too* large.

2.1.2 Scaling a general-form modulus

Rather than *selecting* a special-form modulus N outright, some alternative approaches attempt to *construct* and use $N' = s \cdot N$, i.e., the product of an existing modulus N and scaling factor s; the result, termed a *scaled modulus*, exhibits the special-form required and thereby supports associated optimisations. Note that using N' rather than N potentially implies a larger modulus. As a result, use of a small enough s such that $2^{w \cdot (n-1)} \leq s \cdot N < 2^{w \cdot n}$, i.e., N' and N have the same number of

radix-2^w words, might typically be preferred. Doing so avoids increasing the loop bound in line 2 of Algorithm 1, for example, thereby maximising the value of using N' by avoiding any additional overhead.

Quisquater multiplication [49], first presented at the EUROCRYPT'90 rump session, uses an instance of this approach; we note that Walter [59] independently developed a similar approach around the same time. Following the presentation in [32], consider that the quotient and remainder stemming from division of some X by N can be computed as $Q = \lfloor X/N \rfloor$ and $R = X - Q \cdot N = X - \lfloor X/N \rfloor \cdot N$ respectively. The integer division required to compute Q is (relatively) inefficient, so use of an *approximate* quotient $\hat{Q}$ can be more attractive when a) $\hat{Q}$ can be computed more efficiently than Q, and b) any error resulting from use of $\hat{Q}$ vs. Q can be corrected via a small number of efficient sub-steps. Quisquater observed that Q is lower-bounded by

$$\hat{Q} = \lfloor \frac{X}{2^c \cdot 2^{w \cdot n}} \rfloor \cdot \lfloor \frac{2^c \cdot 2^{w \cdot n}}{N} \rfloor$$

and yields an approximate remainder $\hat{R} = X - \hat{Q} \cdot N$. If one selects a c such that $s = \lfloor (2^c \cdot 2^{w \cdot n})/N \rfloor$ then pre-computes and uses the modulus $N' = s \cdot N$, the required remainder can be computed as $\hat{R} = X - \lfloor X/2^{c+(w \cdot n)} \rfloor \cdot N'$. therefore; this is (more) efficient, because the integer division involved can be replaced with an appropriate shift. Note that, by construction, the c MSBs of the therefore special-form modulus N' are equal to 1, but we did not need to *select* the existing, general modulus N with that property.

Hars [30, Section 5.4] describes an approach he terms *tail tailoring*, set within the context of Montgomery multiplication. The idea is to set $s = \mu = -N^{-1} \pmod{2^w}$, i.e., the pre-computed value related to N: doing so implies

$$N' = s \cdot N \equiv -N^{-1} \cdot n_0 \equiv -n_0^{-1} \cdot n_0 \equiv -1 \pmod{2^w},$$

recalling that n_0 denotes the 0-th or least-significant word of N, and so the pre-computed value related to N' is $\mu' = -N'^{-1} \equiv 1 \pmod{2^w}$ as required to enable the strategy outlined above. Note that, by construction, the w LSBs of the therefore special-form modulus N' are equal to 1, but we did not need to *select* the existing, general modulus N with that property.

2.2 Capitalising on parallelism

It should be obvious that, in the period since publication of [52], almost any platform one might expect RSA to be implemented on has vastly improved with respect to computational, communication, and storage capabilities. It is interesting to highlight a symbiotic relationship between study of the algorithms for modular arithmetic, and platforms that any associated implementation is then developed. A common theme is that algorithms and platforms co-evolve to some extent, so that, e.g., an improvement in the latter might be capitalised on by the former *and* vice versa.

Support for parallel computation has, in particular, evolved from a once niche to now commodity feature in most platforms; it represents a central means of addressing the limitations on scaling (e.g., clock frequency) that stem from Moore's Law [45]. Flynn's Taxonomy [24] offers a structured way to reason about the use of parallelism, describing instances as either single (i.e., scalar) or multiple (i.e., parallel) along two dimensions, namely instructions and data: the four classes are Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD), and Multiple Instruction Multiple Data (MIMD). Given the computational demands of modular arithmetic, it is unsurprising that one can identify work attempting to capitalise on any class outlined above as a way to deliver improved efficiency. However, for various reasons, SIMD is arguably the most interesting. For example, it has wide-scale support through instances of the Intel MMX/SSE/AVX, ARM NEON, PowerPC AltiVec, and AMD 3DNow! families, *and* aligns with genuine *vector*-like support from, for example ARM SVE, and the RISC-V V extension. In addition, it forces lower-level relationship between algorithms and platform, in the sense the ISA dictates the exact form of SIMD support.

Algorithm 2 Bos et. al [13, Algorithm 2]: a *parallel* radix-2^w interleaved (SIMD-friendly) Montgomery multiplication algorithm.

Input: A, B, M, and μ, such that $A = \sum_{i=0}^{n-1} a_i 2^{wi}$, $B = \sum_{i=0}^{n-1} b_i 2^{wi}$, $M = \sum_{i=0}^{n-1} m_i 2^{wi}$, $0 \leq A, B < M$, $2^{w \cdot (n-1)} \leq M < 2^{w \cdot n}$, $2 \nmid M$, $\mu = M^{-1} \bmod 2^w$.
Output: $C \equiv A \cdot B \cdot 2^{-w \cdot n} \bmod M$ such that $0 \leq C < M$.

1: **parallel**
2: **thread**
3: $D = 0$, **i.e.,** $d_i = 0$ **for** $0 \leq i < n$
4: **for** $j = 0$ **to** $n - 1$ **do**
5: $q \leftarrow ((\mu \cdot b_0) \cdot a_j + \mu \cdot (d_0 - e_0)) \bmod 2^w$
6: $t_0 \leftarrow a_j \cdot b_0 + d_0$, $t_0 \leftarrow \lfloor t_0 / 2^w \rfloor$
7: **for** $i = 1$ **to** $n - 1$ **do**
8: $p_0 \leftarrow a_j \cdot b_i + t_0 + d_i$, $t_0 \leftarrow \lfloor p_0 / 2^w \rfloor$, $d_{i-1} \leftarrow p_0 \bmod 2^w$
9: **end for**
10: $d_{n-1} \leftarrow t_0$
11: **end for**
12: **end thread**
13: **thread**
14: $E = 0$, **i.e.,** $e_i = 0$ **for** $0 \leq i < n$
15: **for** $j = 0$ **to** $n - 1$ **do**
16: $q \leftarrow ((\mu \cdot b_0) \cdot a_j + \mu \cdot (d_0 - e_0)) \bmod 2^w$
17: $t_1 \leftarrow q \cdot m_0 + e_0$, $t_1 \leftarrow \lfloor t_1 / 2^w \rfloor$
18: **for** $i = 1$ **to** $n - 1$ **do**
19: $p_1 \leftarrow q \cdot m_i + t_1 + e_i$, $t_1 \leftarrow \lfloor p_1 / 2^w \rfloor$, $e_{i-1} \leftarrow p_1 \bmod 2^w$
20: **end for**
21: $e_{n-1} \leftarrow t_1$
22: **end for**
23: **end thread**
24: **end parallel**
25: $C = D - E$
26: **if** $C < 0$ **then**
27: $C \leftarrow C + M$
28: **end if**
29: **return** C

2.2.1 Intra-multiplication parallelism

Bos et. al [13] adopt an *intra*-multiplication (i.e., *within* each multiplication) approach to parallelisation, splitting Algorithm 1 on page 2 into two threads, which can then be computed in parallel. We continue to use this term per [13], but note that lanes might be a better choice: the idea is to compute in two SIMD-based lanes within one execution context, not two execution contexts (i.e., a multi-threaded implementation). Their approach is based on two main ideas. First, lines 3, 4, and 5 of Algorithm 1 capture $1 \times n$, 1×1, and $1 \times n$ word multiplications respectively; these constitute the computational core, but are difficult to parallelise due to dependencies between them. Notice, however, that computation of $q = \mu C \bmod 2^w$ requires c_0 alone: this fact implies it is possible to compute the $1 \times n$ word multiplications in parallel, *if* one duplicates the computation of c_0 in *each* thread so as to eliminate the problematic dependency. Second, rather than use the Montgomery constant $\mu = -M^{-1} \bmod 2^w$ as is, the sign is flipped to yield $\mu = M^{-1} \bmod 2^w$ instead. Doing so yields two outcomes, namely that 1) both D and E are bounded by M, meaning they can be represented in n (vs. $n + 1$) words; this fact simplifies management of carries, and 2) the conditional final subtraction in Algorithm 1 becomes a conditional final addition.

Algorithm 2 reproduces [13, Algorithm 2], thus capturing the approach for completeness.

Notice that the threads detailed in lines 2 to 12 and 13 to 23 adhere to a SIMD computational model, in the sense they perform the same operations (e.g., line 8 vs. 19) with different data (resp. D vs. E). The only caveat is perhaps lines 5 and 16, which compute q. Since *both* require d_0 and e_0, some synchronisation is required; in reality, it is likely easier to compute q sequentially then distribute the result to both threads. The fact there are two threads somewhat specialises the approach to ISAs with support for 2-way (or 2-lane) SIMD; the authors focus on $w = 32$ specifically, in line with concrete ISAs providing such support. Compared with a sequential 32-bit implementation, [13, Section 4] reports improvement by a factor of 1.68 to 1.76 using a 2048-bit modulus on platforms enabled with the Intel SSE (e.g., Intel Xeon) and ARM NEON (e.g., ARM Cortex-A9) ISAs. However, vs. a sequential 64-bit implementation the results are not as positive: the parallel approach is limited by the form of SIMD supported, in the sense the ISAs allow (32×32)-bit multiplication only. Constraints imposed by, e.g., non-orthogonality of the ISA are a common challenge for SIMD-based implementations. For example, in early work Acar [1, Section 5.4.1] noted the difficulty of using MMX for modular arithmetic due to a lack of suitable unsigned multiplication instructions. This issue has arguably improved over time, as ISAs have evolved away from their media-oriented origins and toward support for general-purpose workloads. Either way, use of *intra*-multiplication parallelism is advantageous in the sense that improvement could be harnessed by any workload based on modular multiplication: use of modular exponentiation in RSA is one example, but, equally, others can (transparently) benefit.

2.2.2 Inter-multiplication parallelism

Within the context of cryptographic implementation, the term bit-slicing is normally attributed to Biham [10]: it refers to a non-standard *representation* of data, plus a non-standard *implementation* of functions that operate on instances of said representation. Consider a w-bit word x, where x_i denotes the i-th bit for $0 \leq i < w$. Computing the result of operations on such words, e.g., $x \oplus y$, the XOR of words x and y, is rendered efficient by native support in the underlying processor. However, computing the result of $x_i \oplus x_j$, i.e., the XOR of bits *within* x, is more difficult due to a lack of the same (native) support. Where such operations are common, the use of bit-slicing can be attractive: it transforms such an x into w slices, say $\hat{x}[i]$ for $0 \leq i < w$, such that $\hat{x}[i]_k = x_i$ (i.e., the i-th bit of x) for some k. Put another way, the i-th bit of x is placed at the k-th index within the i-th slice. Visually, this transformation is described by

$$x = \langle x_0, x_1, \ldots, x_{w-1} \rangle \quad \mapsto \quad \left\{ \begin{array}{lcl} \hat{x}[0] & = & \langle \ \ldots, \ x_0, \ \ldots \ \rangle \\ \hat{x}[1] & = & \langle \ \ldots, \ x_1, \ \ldots \ \rangle \\ & \vdots & \\ \hat{x}[w-1] & = & \langle \ \ldots, \ x_{w-1}, \ \ldots \ \rangle \end{array} \right.$$

Under such a representation, the original operation can again be efficient: we can (natively) compute $\hat{x}[i] \oplus \hat{x}[j]$, because x_i and x_j are at the same index in those slices. More generally, any function previously used as $r = f(x)$ must be transformed into an alternative $\hat{r} = \hat{f}(\hat{x})$ in order to process bit-sliced operands. This implies two disadvantages: 1) there is an overhead related to the conversion of x into $\hat{x}$ and $\hat{r}$ into r, and 2) many operations must be translated into a "software circuit", composed of Boolean operations on the slices, within $\hat{f}$; although one can (natively) compute $x + y$, for example, the same is not true of $\hat{x}$ and $\hat{y}$. Crucially, however, if each slice is itself represented as a w-bit word, then it is possible to compute w instances of $\hat{f}$ in *parallel* on suitably packed $\hat{x}$. A common analogy is that of bit-slicing transforming the w-bit, 1-way scalar processor into a 1-bit, w-way SIMD processor, thus yielding upto a w-fold improvement which acts to compensate for the disadvantages.

An analogous technique can be applied to (modular) integer arithmetic using a radix-2^w representation: the idea is again to slice an $X \in \mathbb{Z}$ into n slices, say $\hat{X}[i]$ for $0 \leq i < n$, such that

$\hat{X}[i]_k = x_i$ (i.e., the i-th word of X) for some k. Visually, this transformation is described by

$$X = \langle x_0, x_1, \ldots, x_{n-1} \rangle \quad \mapsto \quad \left\{ \begin{array}{lcl} \hat{X}[0] & = & \langle \ \ldots, \ x_0, \ \ldots \ \rangle \\ \hat{X}[1] & = & \langle \ \ldots, \ x_1, \ \ldots \ \rangle \\ & \vdots & \\ \hat{X}[n-1] & = & \langle \ \ldots, \ x_{n-1}, \ \ldots \ \rangle \end{array} \right.$$

As with bit-slicing this can make it easier to combine words within X, say x_i and x_j, e.g., to deal with carries between words. This technique, termed word-slicing, seems to have independent origins from bit-slicing itself. More specifically, Montgomery [44] originally observed that, the vectorisation of modular arithmetic "horizontally" (or intra-operation) was more difficult than "vertically" (or inter-operation) on a vector-based Cray Y-MP. To solve this problem he adopted word-slicing, vectorising an ECM [41] implementation "vertically" to allow multiple trials in parallel (vs. the "horizontal" parallelisation of one trial). Page and Smart [46] rediscover the technique in adopting an *inter*-multiplication approach to parallelisation. Their idea is to support parallel computation of $R[k] = X[k]^e \pmod{N[k]}$ for $0 \leq k < 4$, i.e., four exponentiations using *different* bases and moduli but the *same* exponent; note that use of the same exponent implies uniform control-flow within each exponentiation, i.e., they follow SIMD-style computation. Set within the context of RSA this permits a form of "batch" encryption or decryption, e.g., by a server dealing with multiple clients. Implementing the parallel exponentiation reduces to implementing parallel Montgomery multiplication, namely Algorithm 1, through use of word-slicing: one simply slices and packs words in

$$\begin{array}{lcl} R[k] & = & \langle \ r[k]_0, \ r[k]_1, \ \ldots, \ r[k]_{n-1} \ \rangle \\ N[k] & = & \langle \ n[k]_0, \ n[k]_1, \ \ldots, \ n[k]_{n-1} \ \rangle \\ X[k] & = & \langle \ x[k]_0, \ x[k]_1, \ \ldots, \ x[k]_{n-1} \ \rangle \end{array}$$

for $0 \leq k < 4$ to yield

$$\begin{array}{lcl} \hat{R}[i] & = & \langle \ r[0]_i, \ r[1]_i, \ r[2]_i, \ r[3]_i \ \rangle \\ \hat{X}[i] & = & \langle \ x[0]_i, \ x[1]_i, \ x[2]_i, \ x[3]_i \ \rangle \\ \hat{N}[i] & = & \langle \ n[0]_i, \ n[1]_i, \ n[2]_i, \ n[3]_i \ \rangle \end{array}$$

for $0 \leq i < n$, then implements a suitable $\hat{f}$ such that $\hat{R} = \hat{f}(\hat{X}, e, \hat{N})$ computes the required exponentiations. The only arithmetic complication is the conditional final subtraction step in lines 7 to 9 of Algorithm 1, as used to produce the least residue modulo N such that $0 \leq C < N$ as output; without it, the output would satisfy $0 \leq C < 2 \cdot N$. As observed by Walter [60, 61] (and others [28, 27]), where Algorithm 1 is used iteratively (where the output is reused as a subsequent input, e.g., in an exponentiation) the subtraction step can be eliminated whenever $4 \cdot N < 2^{wn}$ since then all input and output to the Montgomery multiplication are bounded by $2 \cdot N$ and represented in a redundant Montgomery form. Having removed the conditional subtraction, a Montgomery multiplication can be computed with no data-dependent control-flow; the (minor) trade-off is a requirement to cater for 1-word larger operands when the length of the modulus is close to a multiple of w (which for standard RSA parameters, it will be). Compared with a sequential 32-bit implementation, [46, Section 3.3] reports improvement by close to a factor of 2 using a 2048-bit modulus on platforms enabled with the Intel SSE (Intel Pentium 4) ISA. Of course, realising the improvement in practice assumes a usable batch of exponentiations is available; where such a batch cannot be guaranteed, use of an *intra*-multiplication approach [13] could be a more sensible use of the computational resources.

2.3 Dealing with large moduli

Set against the context of contemporary factoring algorithms (as discussed in [52, Section IX]), Rivest, Shamir, and Adleman "recommend[ed] using 100-digit (decimal) prime numbers p and q,

so that [N] has 200 digits" [52, Section VII] to ensure the security of RSA; this means a 665-bit N. However, the technology landscape has obviously evolved since then. This has meant improvement in factoring algorithms, their implementation, the platforms they are executed on, and therefore, ultimately, their efficiency: all these factors have contributed to a significant increase in the length of plausibly factorable moduli, and hence recommendations for secure parameterisation.

For example, in 2001 Lenstra and Verheul [38] developed a methodology and analysis of parameter selection for a range of cryptographic constructions, including RSA. They conclude that "RSA keys that are supposed to be secure until 2040 are about three times larger than the popular 1024-bit RSA keys that are currently secure" noting an impact on the efficiency of associated modular arithmetic: use of RSA will be "9 – 27 times slower". ENISA [55, Section 3.6] offer a longer-term perspective, recommending 15360-bit moduli to ensure "security [of RSA] for thirty to fifty years". Resources such as

https://www.keylength.com

offer a useful summary of recommendations, over time and from different sources. Even viewing such recommendations as approximate, the obvious challenge is how to scale, i.e., how to mitigate the impact of increased moduli lengths on the efficiency of associated modular arithmetic and hence RSA.

2.3.1 "Double-length" arithmetic

Consider the Karatsuba-Ofman [33] technique, which allows the computation of a product $R = X \cdot Y$ by decomposition. For n-bit integers X and Y, assuming for simplicity that n is even, it decomposes

$$\begin{aligned} X &= X_1 \cdot 2^{n/2} + X_0 \\ Y &= Y_1 \cdot 2^{n/2} + Y_0 \end{aligned}$$

where X_i and Y_i are then $(n/2)$-bit integers, then computes

$$R = R_2 \cdot 2^n + R_1 \cdot 2^{n/2} + R_0$$

where

$$\begin{aligned} R_0 &= T_0 \\ R_1 &= T_1 - T_0 - T_2 \\ R_2 &= T_2 \end{aligned}$$

$$\begin{aligned} T_0 &= X_0 \cdot Y_0 \\ T_1 &= (X_0 + X_1) \cdot (Y_0 + Y_1) \\ T_2 &= X_1 \cdot Y_1 \end{aligned}$$

Put another way, this technique realises an n-bit integer multiplication using a) three $(n/2)$-bit multiplications, plus b) some auxiliary additions and subtractions. In the context of hardware implementation, for example, this is attractive because it enables various trade-offs including between area and latency: it implies reduced area of a $(n/2)$-bit vs. n-bit multiplier, as a trade-off against increased latency with respect to their reuse when computing T_0, then T_1, then T_2.

One might consider scaling RSA using a conceptually similar technique: if we *have* efficient modular arithmetic for an n-bit modulus (e.g., in the form of dedicated hardware, or co-processor), the idea would be to somehow leverage it in delivering what we *want*, i.e., arithmetic for a $(2 \cdot n)$-bit or "double-length" modulus. Paillier [47, Section 1] succinctly outlines two research challenges of this type, namely "[h]ow to optimally implement nk-bit modular operations using k-bit modular operations" and, more specifically, "[h]ow to implement an nk-bit modular multiplication using k-bit modular operations with a minimal number of k-bit multiplications", leading to various associated work (see, e.g., [19, 63]). As a representative example, consider the solution of Fischer and Seifert [23] which assumes an API that provides an operation

$$(Q, R) = \texttt{MultModDiv}(X, Y, M)$$

Algorithm 3 Fischer and Seifert [23, Section 3.1]: "basic doubling" algorithm.

Input: $\left.\begin{array}{l} N = N_t \cdot 2^n + N_b \\ X = X_t \cdot 2^n + X_b \\ Y = Y_t \cdot 2^n + Y_b \end{array}\right\}$ such that $0 \leq N_b, X_b, Y_b < 2^n$

Output: $R = X \cdot Y \pmod{N}$

1: $(Q_1, R_1) \leftarrow \texttt{MultModDiv}(Y_t, 2^n, N_t)$
2: $(Q_2, R_2) \leftarrow \texttt{MultModDiv}(Q_1, N_b, 2^n)$
3: $(Q_3, R_3) \leftarrow \texttt{MultModDiv}(X_t, R_1 - Q_2 + Y_b, N_t)$
4: $(Q_4, R_4) \leftarrow \texttt{MultModDiv}(X_b, Y_t, N_t)$
5: $(Q_5, R_5) \leftarrow \texttt{MultModDiv}(Q_3 + Q_4, N_b, 2^n)$
6: $(Q_6, R_6) \leftarrow \texttt{MultModDiv}(X_t, R_2, 2^n)$
7: $(Q_7, R_7) \leftarrow \texttt{MultModDiv}(X_b, Y_b, 2^n)$
8: $Q \leftarrow R_3 + R_4 - Q_5 - Q_6 + Q_7$
9: $R \leftarrow R_7 - R_6 - R_5$
10: **return** $Q \cdot 2^n + R \pmod{N}$

for an n-bit M, where

$$\begin{array}{rcl} Q & = & (X \cdot Y) \bmod M \\ R & = & \lfloor (X \cdot Y)/M \rfloor \end{array}$$

The simpler of two algorithms is presented in [23, Section 3.1]: it realises a $(2 \cdot n)$-bit multiplication modulo N, by using seven invocations of `MultModDiv` (noting that some can be executed in parallel, if/when possible), plus some auxiliary additions and subtractions. We reproduce the approach in Algorithm 3 for completeness, but omit the proof of correctness provided in detail by [23, Section 3.1].

2.3.2 "Approximate-then-correct" arithmetic

Using the long-term ENISA [55, Section 3.6] recommendation as motivation, Bentahar and Smart [5] explore the efficiency of algorithms for arithmetic modulo a 15,360-bit N, via both analytical and experimental approaches: their premise is that asymptotically efficient yet concretely unsuitable algorithms, for smaller moduli, become "in scope" for such larger moduli.

[5, Section 2.4] outlines the use of "wooping", a technique attributed to Bos [12, Chapter 6] (see also [22, Section 15.1.1]) as a means of error correction. Imagine one computes the integer product $R = X \cdot Y$. To (probabilistically) verify whether R is correct, one selects a random prime p, computes

$$\begin{array}{rcll} \hat{X} & = & X & \pmod{p} \\ \hat{Y} & = & Y & \pmod{p} \\ \hat{R} & = & \hat{X} \cdot \hat{Y} & \pmod{p} \end{array}$$

then tests whether $R \stackrel{?}{=} \hat{R} \pmod{p}$. If the equality does not hold we infer R is incorrect; if the equality holds, the probability of a false-positive is $1/p$. $\hat{R}$ can be viewed as an arithmetic checksum, which can a) be efficiently computed if p is small (e.g., a single word, meaning w bits; this is particularly true when also using special-form p), and b) offers a low false-positive probability if p is large *enough*, or multiple such checksums are used. This technique is then used in [5, Section 3], in order to yield efficient algorithms for Montgomery [43] and Barrett [3] reduction. The central idea is to approximate certain *full* products using (more efficient) half products, then correct said approximations via wooping.

3 Arithmetic for ECC

Besides RSA (see Section 2 on page 2), the other popular approach to realize public-key cryptography is based on the algebraic structure of elliptic curves over finite fields. This *elliptic curve cryptography* (ECC) [36, 42] enjoys increasing popularity since its invention in the mid 1980s. This became the preferred alternative to RSA due to the attractiveness of smaller key-sizes [39, 40].

As opposed to RSA, the cryptographic standard includes many parameters which determine the exact choice of curve used and over which finite field. Since the initial standardization of elliptic curve cryptography [57] there has been significant progress: this includes taking into account various new types of attacks (such as side-channel attacks) and performance improvements. This latter area includes using different curve models, which lowers the total cost of elliptic curve group operations. However, all the curve operations have to be implemented using a sequence of operations in the underlying finite field. There have been advances in the choice of the shape of these primes in order to increase the performance of the modular arithmetic.

In this Section we will describe the choices made for the prime shapes in the ECC standard by the National Institute of Standards and Technology (NIST) [56], why these primes result in particularly efficient modular reduction implementations and what caused the choice of slightly different prime shapes with the new elliptic curves.

3.1 Generalized Mersenne numbers

It is well known that reduction modulo Mersenne primes is very efficient. A Mersenne prime is a prime number which is one less than a power of two: $2^x - 1$. When computing $a \cdot b$ modulo $2^x - 1$ with $0 \leq a, b < 2^x - 1$, one can compute the reduction without multiplications as

$$c = a \cdot b = c_1 \cdot 2^x + c_0 \equiv c_1 + c_0 \pmod{2^x - 1}$$

where $0 \leq c_0, c_1 < 2^x$, since $2^x \equiv 1 \bmod 2^x - 1$. However, for even more practical convenience, one would like x to be close to a multiple of 32 or 64 since this matches (a multiple of) the word size of virtually all modern computer architectures.

Unfortunately, there are not many Mersenne primes in the range where this is of interest for elliptic curve cryptography. For example, when looking at the range $100 < x < 1000$ only four such values are available: $x \in \{107, 127, 521, 607\}$. The modulus $2^{127} - 1$ has been proposed for usage with hyperelliptic curve cryptography in genus 2 (cf. [8, 25, 15]) where it offers sufficient security while $2^{521} - 1$ is used in the NIST standard for elliptic curve (genus 1) cryptography.

One direction to generalize this idea is to use prime moduli of the form $2^x - c$ where c is small enough to fit in a computer word. These are sometimes referred to as *Crandall numbers* [20] and will be explored in more detail in Section 3.2 on page 13. Another direction to generalize was studied by Solinas in [56] and later adopted by NIST. A potential reason to go for these *generalized Mersenne numbers* over Crandall numbers might be patent-related (cf. [20]).

Solinas studied efficient reductions, e.g. reduction which do not require multiplications, modulo polynomials of the form

$$f(t) = t^d + \sum_{i=0}^{d-1} c_i \cdot t^i \tag{2}$$

where $c_i \in \{-1, 0, 1\}$. For selected integers k and d these techniques can then be used for efficient reduction modulo $f(2^k)$ as will be shown in more detail in this section. The five selected generalized Mersenne primes of this shape which are specified in the NIST standard are

$$\begin{aligned}
p_{192} &= 2^{192} - 2^{64} - 1 \\
p_{224} &= 2^{224} - 2^{96} + 1 \\
p_{256} &= 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 \\
p_{384} &= 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1 \\
p_{521} &= 2^{521} - 1.
\end{aligned}$$

It should be noted that for the first four generalized Mersenne primes all the exponents are a multiple of 32. This makes implementations on 32-bit platforms significantly easier and faster. The exception is for the Mersenne prime p_{521}; here the choice was clearly made to go for a Mersenne prime instead of a generalized Mersenne prime such as $2^{512} - 2^{32} - 1$, $2^{512} - 2^{32} + 1$, or $2^{512} - 2^{288} + 1$.

3.1.1 Using the Prime Shape Directly

Let us give an example how this efficient modular reduction works. We will use the modulus p_{256} as an example since this is arguably the most frequently used prime out of these five since this targets the 128-bit security level when used with an appropriate elliptic curve for usage in cryptography. Similar to the approach used for Mersenne primes one writes large powers of two (where the exponent is ≥ 256) in terms of smaller powers of two. For example, $2^{256} \equiv 2^{224} - 2^{192} - 2^{96} + 1$ by the definition of p_{256}. For larger exponents this can be done similarly and one obtains

$$\begin{aligned}
2^{256} &\equiv 2^{224} - 2^{192} - 2^{96} + 1, \\
2^{288} &\equiv -2^{192} - 2^{128} - 2^{96} + 2^{32} + 1, \\
2^{320} &\equiv -2^{224} - 2^{160} - 2^{128} + 2^{64} + 2^{32}, \\
2^{352} &\equiv -2^{224} - 2^{160} + 2 \cdot 2^{96} + 2^{64} - 1, \\
2^{384} &\equiv -2^{224} + 2 \cdot 2^{128} + 2 \cdot 2^{96} - 2^{32} - 1, \\
2^{416} &\equiv -2^{224} + 2^{192} + 2 \cdot 2^{160} + 2 \cdot 2^{128} + 2^{96} - 2^{64} - 2^{32} - 1, \\
2^{448} &\equiv 3 \cdot 2^{192} + 2 \cdot 2^{160} + 2^{128} - 2^{64} - 2^{32} - 1, \\
2^{480} &\equiv 3 \cdot 2^{224} + 2 \cdot 2^{192} + 2^{160} - 2^{96} - 2^{64} - 2^{32}.
\end{aligned}$$

Hence, after an initial multiplication $c = a \cdot b$ for $0 \leq a, b < p_{256}$ the modular reduction can be carried out efficiently by substituting the powers of two in $c = \sum_{i=0}^{15} c_i 2^{32i}$ with the congruent values from above. When grouping terms together things boil down to

$$c = a \cdot b = \sum_{i=0}^{15} c_i 2^{32i} \equiv s_1 + 2s_2 + 2s_3 + s_4 + s_5 - s_6 - s_7 - s_8 - s_9 \bmod p_{256}$$

where the 32-bit coefficients of the s_i are defined in terms of the coefficients c as

	2^{224}	2^{192}	2^{160}	2^{128}	2^{96}	2^{64}	2^{32}	2^{0}
s_1	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0
s_2	c_{15}	c_{14}	c_{13}	c_{12}	c_{11}	0	0	0
s_3	0	c_{15}	c_{14}	c_{13}	c_{12}	0	0	0
s_4	c_{15}	c_{14}	0	0	0	c_{10}	c_9	c_8
s_5	c_8	c_{13}	c_{15}	c_{14}	c_{13}	c_{11}	c_{10}	c_9
s_6	c_{10}	c_8	0	0	0	c_{13}	c_{12}	c_{11}
s_7	c_{11}	c_9	0	0	c_{15}	c_{14}	c_{13}	c_{12}
s_8	c_{12}	0	c_{10}	c_9	c_8	c_{15}	c_{14}	c_{13}
s_9	c_{13}	0	c_{11}	c_{10}	c_9	0	c_{15}	c_{14}

Computing the reduction modulo p_{256} can be done using six modular additions and four modular subtractions. Computing the modular addition $c = a + b \bmod p_{256}$ in constant running time (see Section on page 1) can be done at the cost of approximately one addition and one subtraction using

$$c_1 \leftarrow a + b, \quad c_2 \leftarrow c_1 - p_{256}, \quad c \leftarrow \text{select}(\lfloor c_1 / 2^{256} \rfloor, c_1, c_2)$$

where the function $\text{select}(x, a, b)$ return a if $x = 0$ or b otherwise. Such a selection can be implemented efficiently in constant time by masking out the correct results. Ergo, the total cost of a constant-time modular reduction is ten 256-bit additions and ten 256-bit subtractions.

3.1.2 Using Montgomery Reduction

Another approach to perform arithmetic modulo generalized Mersenne numbers is using Montgomery arithmetic (see Section 1 on page 1). Recall that radix-2^w Montgomery reduction requires a multiplication with the pre-computed Montgomery constant $\mu = -N^{-1} \bmod 2^w$ and a multiplication with the modulus N. Similar to the observation made in Section 2.1.2 on page 4 in the setting of RSA the multiplication with μ can be avoided whenever $N \equiv \pm 1 \bmod 2^w$ since then $\mu \equiv \mp 1 \bmod 2^w$. This observation has been made and used in various cryptographic applications before, see for example Lenstra [37], Acar and Shumow [2], Knežević, Vercauteren, and Verbauwhede [34], Hamburg [29], and Bos, Costello, Hisil, and Lauter [15, 16].

Turning to our running example when using p_{256} this multiplication can be omitted since $-p_{256}^{-1} \bmod 2^x = 1$ for all positive integers $x \leq 96$. Let us present an approach where one uses a Montgomery-radix of 2^{64} for a 32-bit platform when computing $c = A \cdot B = \sum_{i=0}^{3} a_i 2^{64i} \cdot B \bmod p_{256}$. After computing and accumulating the product of a_i with B as $c = c + a_i \cdot B$ the Montgomery reduction can be simplified as

$$\begin{aligned} c &= (c + p_{256} \cdot (c_1 \cdot 2^{32} + c_0))/2^{64} \\ &= \left(c + \left(c_0 \cdot 2^{256} - c_0 \cdot 2^{224} + c_0 \cdot 2^{192} + c_0 \cdot 2^{96} - c_0\right) + \right. \\ &\quad \left. \left(c_1 \cdot 2^{288} - c_1 \cdot 2^{256} + c_1 \cdot 2^{224} + c_1 \cdot 2^{128} - c_1 \cdot 2^{32}\right)\right)/2^{64} \\ &= \left(\sum_{i=2}^{9} c_i \cdot 2^{32(i-2)}\right) - \left(c_1 \cdot 2^{192} + c_0 \cdot 2^{160}\right) + \\ &\quad \left(c_1 \cdot 2^{224} + c_0 \cdot 2^{192} + c_1 \cdot 2^{160} + c_0 \cdot 2^{128} + c_1 \cdot 2^{64} + c_0 \cdot 2^{32}\right) \end{aligned}$$

when using the special shape of p_{256}. This process reduces the size of the result by 64 bits and needs to be performed four times in total to compute a full Montgomery reduction. This results in the following interleaved Montgomery multiplication routine for p_{256} on a 32-bit platform:

$c \leftarrow 0$
for $i = 0$ to 3 **do**
  $c \leftarrow c + a_i \cdot B$
  $t_1 \leftarrow c_1 \cdot 2^{224} + c_0 \cdot 2^{192} + c_1 \cdot 2^{160} + c_0 \cdot 2^{128} + c_1 \cdot 2^{64} + c_0 \cdot 2^{32}$
  $t_2 \leftarrow c_1 \cdot 2^{192} + c_0 \cdot 2^{160}$
  $c \leftarrow \left\lfloor \frac{c}{2^{64}} \right\rfloor + t_1 - t_2$
end for

Hence, the Montgomery reduction modulo p_{256} takes only four 256-bit additions and four 256-bit subtractions in total. When reduction between zero and p_{256} is required a conditional subtraction needs to be computed which can be done using one additional 256-bit addition and one additional 256-bit subtraction. This approach is more efficient compared to the direct approach described in the previous subsection and allows for a significantly simpler implementation.

3.2 Arithmetic for Curve25519

Since the standardization of the NIST curves, together with the prime used to define the finite field, a significant amount of progress has been made in the field of elliptic curve cryptography with respect to the performance and security properties. This lead Bernstein to introduce an alternative elliptic curve for usage in public-key cryptography [7]. This curve is denoted *Curve25519*, referring to the shape of the prime used to define the finite field: $2^{255} - 19$. From a modular arithmetic point of view this approach deviates from the generalized Mersenne approach used by NIST. Bernstein proposes to use a Crandall number [20] of the form $2^x - c$ where c is small compared to the word size used instead.

Bernstein proposes to represent integers modulo $2^{255} - 19$ as elements of the ring of polynomials $\sum_{i=0}^{9} u_i x^i$, where u_i is an integer multiple of $2^{\lceil 25.5i \rceil}$ such that $a_i/2^{\lceil 25.5i \rceil} \in \{-2^{25}, -2^{25} + 1, \ldots, -1, 0, 1, \ldots, 2^{25} - 1, 2^{25}\}$ are a reduced-degree reduced-coefficient polynomial and represent elements of $\mathbb{Z}/(2^{255} - 19)\mathbb{Z}$ where each polynomial represents its value at one. Hence, the idea is to represent a 255-bit integer using ten 26-bit pieces. This approach is motivated by using the fast floating-point operations available on most modern processors.

Although this floating point approach is of interest by itself, it is not the preferred approach used by many of the implementations in cryptographic libraries. In practice, either a representation which allows a non-unique representation of the elements to avoid carry-propagation is used or a redundant representation. Both approaches are outlined below.

One common approach to implement efficient multi-precision arithmetic is to use a non-unique representation where the coefficients have sufficient space to grow without the need for carry propagation. This allows efficient accumelation of results during the multiplication and reduction step. For example, in the implementation approach of EdDSA [9], which targets 64-bit Intel and AMD CPUs, a radix-2^{64} can cause bottleneck when dealing with carries. Therefore an element $x \in \mathbb{Z}/(2^{255} - 19)\mathbb{Z}$ is represented as $x = \sum_{i=0}^{4} x_i 2^{51i}$. A unique representation would require that each $0 \leq x_i < 2^{51}$, however, in practice the coefficients x_i are stored in 64-bit computer words which avoid carry propagation. The idea is that after multiplication $c = a \cdot b$ for $0 \leq a, b < 2^{255} - 19$ the integer $c = \sum_{i=0}^{9} c_i 2^{51i}$ can be efficiently reduced by computing

$$(x_i + 19x_{i+5})\, 2^{51i}, \quad \text{for } 0 \leq i \leq 4.$$

Moreover, when the coefficients x_i are $0 \leq x_i < 2^{51+\delta}$ for a positive integer $\delta > 0$ then $(x_i + 19x_{i+5})\, 2^{51i} < 2^{56+\delta}$. In practice, the multiplication accepts inputs with each limb having up to 54 bits (i.e., $\delta = 3$). The disadvantage of this approach is that comparisons in this non-unique representation become more cumbersome and, more importantly, more registers are needed to represent an element.

When the number of available registers is limited it might be more efficient to work with a redundant representation and compute with elements from $\mathbb{Z}/(2(2^{255} - 19))\mathbb{Z} = \mathbb{Z}/(2^{256} - 38)\mathbb{Z}$. For example, Düll, Haase, Hinterwälder, Hutter, Paar, Sánchez and Schwabe outline the most efficient approach for the embedded platforms AVR ATmega (8-bit), MSP430X (16-bit), and the ARM Cortex M0 (32-bit) in [21]. This approach uses the "straight-forward" approach as also considered by Bos in [14]. Given two integers a, b such that $0 \leq a, b < 2^{256} - 38$ then first compute the multiplication $c = a \cdot b$. An initial reduction step can be computed with

$$c = c_1 2^{256} + c_0 \equiv c_1 38 + c_0 \bmod 2^{256} - 38$$

using a single 6×256 bits multiplication where $0 \leq c_0, c_1 < 2^{256}$. The resulting integer $d = c_1 38 + c_0$ can be further reduced similarly $d = d_1 2^{256} + d_0 \equiv d_1 38 + d_0 \bmod 2^{256} - 38$ using a 6×6 bits multiplication. The resulting $d_1 38 + d_0 < 38^2 + 2^{256}$ and a final conditional subtraction is needed to reduce the input to the range $[0, \ldots, 2^{256} - 38 - 1]$.

4 Special Arithmetic

Apart from RSA and ECC there are many more situations in cryptology where efficient modular multiplication is needed. As in the case of RSA and ECC most or all cryptosystems allow specialising parameters in order to speed up their arithmetic. Instead of enumerating all improvements of modular multiplication in special stituations we focus on two examples, namely sloppy reduction [17] which speeds up modular reduction for certain moduli at the expense of correctness, and a method for arithmetic operations modulo Mersenne numbers [18] whose efficiency depends on the available arithmetic instructions.

4.1 Sloppy reduction

As already alluded to at the end of the previous section, modular reduction is quite easy if the modulus is an integer of the form $N = 2^n - r$ (Crandall number) where r is a small positive integer, say, $r^2 + r \leq 2^n$. In this case it is advantageous to extend the range of operands and results from $[0, \ldots, N-1]$ to $[0, \ldots, R-1]$ with $R = 2^n$. The modular reduction of a product $c = a \cdot b$, $0 \leq a, b < R$ can be computed by a repeated application of the following simple reduction step $\mathcal{R}$. For an integer d with $0 \leq d < R^2$, and $d = d_0 + d_1 R$ with $0 \leq d_0, d_1 < R$ define $\mathcal{R}(d) = d_0 + d_1 r$. Equivalently, one has $\mathcal{R}(d) = d - d_1 N$ which implies $\mathcal{R}(d) \equiv d \bmod N$. Moreover, $\mathcal{R}(d) < d$ holds if and only if $d \geq R$. Thus, by applying $\mathcal{R}$ sufficiently often to the product c it will be reduced to an integer below R and a final conditional subtraction reduces it to the range $[0, \ldots, N-1]$ if so desired. It is easy to show (cf. below) that for $r^2 \leq R$ three reduction steps are always sufficient; for $r = 1$ two steps suffice.

Since the time spent in a reduction step is not negligible, the idea of sloppy reduction is to skip the third reduction step and output $\mathcal{R}(\mathcal{R}(d)) \bmod R$ as the modular reduction, leading to the following reduction routine

for $i = 1$ to 2 **do**
  $d_0 \leftarrow d \bmod R$
  $d_1 \leftarrow \lfloor \frac{d}{R} \rfloor$
  $d \leftarrow d_0 + d_1 r$
end for
$d \leftarrow d \bmod R$

This can produce a wrong result so sloppy reduction is only useful if the probablility of an incorrect result is sufficiently low. In the case that the result can be verified by other means the probablility must be low enough such that the cost of verification and possible recomputation does not exceed the saving from sloppy reduction. For example, in cryptanalysis it is often the case that the results of only a small fraction of many computations are used and that these results can be checked quickly. If no verification of the result is possible, sloppy reduction may still be used, e.g., if the probablility of an incorrect result is much smaller than the probablility of errors from other sources (algorithmic or hardware).

Analysis of incorrectness Let d be an integer with $0 \leq d < R^2$. In the following, equivalent conditions for $\mathcal{R}(\mathcal{R}(d)) \geq R$, i.e., failure of sloppy reduction for d, will be derived. Writing $d = x + yN = x' + (y - z)R$ with $0 \leq x < N$ and $0 \leq x' < R$ one gets $x' = x - yr + zR$ and thus $0 \leq z \leq r + 1$ (as $y \leq R + r$, a consequence of $r^2 + r \leq R$). Therefore it follows from $\mathcal{R}(d) = x + zN = (x - zr) + zR$ that

$$\mathcal{R}(\mathcal{R}(d)) = \begin{cases} x & x \geq zr \\ x + N & x < zr \end{cases}$$

holds. Thus sloppy reduction fails if and only if $r \leq x < zr$ holds. In order to express this statement on a more managable quantity than z, write $x = u + vr$ with $0 \leq u < r$; from the above one also has $1 \leq v \leq z - 1 \leq r$. Since $x' = x - yr + zR < R$ implies $yr > x + zR - R \geq x + vR$, one obtains

$$d = x + yN > \frac{xr + xN + vRN}{r} = \frac{R(x + vN)}{r} = \frac{R(u + vR)}{r}.$$

Conversely, this inequality implies that $x < zr$ holds. Since $\frac{R(u+vR)}{r} \geq R^2$ for $v \geq r$, the bound on v can be improved to $1 \leq v < r$ (this also implies $\mathcal{R}(\mathcal{R}(d)) < N + r^2 < 2N$ so that a third reduction step will always result in a correct modular reduction, cf. above).

Summarising, one obtains the following

Fact: Let d be an integer with $0 \leq d < R^2$. Then $\mathcal{R}(\mathcal{R}(d)) \geq R$ if and only if there exist $0 \leq u < r$ and $1 \leq v < r$ such that $d \equiv u + vr \bmod N$ and $d > \frac{R(u+vR)}{r}$ hold.

From these conditions one easily obtains an estimate of the number of $0 \le d < R^2$ with $\mathcal{R}(\mathcal{R}(d)) \ge R$. Indeed, it is

$$\sum_{u=0}^{r-1}\sum_{v=1}^{r-1} \#\{d \mid d \equiv u + vr \bmod N \quad \text{and} \quad \frac{R(u+vR)}{r} < d < R^2\}$$

which can be approximated by

$$\sum_{u=0}^{r-1}\sum_{v=1}^{r-1} \frac{rR^2 - R(u+vR)}{Nr} = \sum_{v=1}^{r-1} \frac{(r-v)R^2}{N} - \sum_{u=0}^{r-1} \frac{(r-1)uR}{Nr} = \frac{R(rR-r+1)(r-1)}{2N}$$

with an error of at most $r(r-1)$. Thus, if sloppy reduction is applied to a randomly chosen number in the interval $[0, R^2-1]$, the probability of an incorrect result is about $\frac{r(r-1)}{2N}$.

However, sloppy reduction is usually used after a multiplication of two uniformly distributed factors in the interval $[0, R-1]$, possibly followed by an addition or a subtraction of a uniformly distributed integer in the same interval, so that the input of sloppy reduction is not uniformly distributed in $[0, R^2-1]$. In these cases the analysis is more difficult; the analysis given below will rely on heuristic assumptions.

Consider first the case $c = a \cdot b + a'$ with $0 \le a, b < R$ and $|a'| < R$. For a rough approximation of the failure probability one discards a' and assumes that the condition $c \equiv u + vr \bmod N$ is met with probability $\frac{1}{N}$ for given u and v. The number of pairs (a, b) satisfying $a \cdot b > \frac{R(u+vR)}{r}$ can be approximated by the integral

$$\int_{\frac{(u+vR)}{r}}^{R} (R - \frac{R(u+vR)}{ar})da = R(R - \frac{(u+vR)}{r}) - \frac{R(u+vR)}{r} \log \frac{rR}{u+vR}. \tag{3}$$

By approximating $u + vR$ with vR and summing over u and v one gets the following estimate for the probability of an incorrect result

$$\frac{1}{NR^2}\sum_{u=0}^{r-1}\sum_{v=1}^{r-1} R(R - \frac{vR}{r}) - \frac{vR^2}{r}\log\frac{r}{v} = \frac{1}{N}(\frac{r(r-1)}{2} - \sum_{v=1}^{r-1} v \log\frac{r}{v}).$$

The case $c = a \cdot b$ becomes more involved since the number and properties of solutions of $ab \equiv u + vr \bmod N$ depend on the greatest common divisor of $u + vr$ and N. For example, if $u + vr$ and N are even, at least one of a and b must be even too.

Given $0 \le u < r$ and $1 \le v < r$ let $w = \gcd(u + vr, N)$ and consider the solutions (a, b) of $ab \equiv u + vr \bmod N$. These solutions satisfy $\gcd(a, N) = \gcd(a, w)$. For a fixed divisor $g \mid w$ the number of residue classes of a modulo N satisfying $\gcd(a, N) = \gcd(a, w) = g$ is $\phi(\frac{N}{g})$ and for each such residue class b is determined uniquely modulo $\frac{N}{g}$. Thus the number of residue classes of pairs (a, b) satisfying $ab \equiv u + vr \bmod N$ is $\sum_{g|w} \phi(\frac{N}{g})g$.

Under the heuristic assumption that this expression divided by N^2 approximates the probability that a pair (a, b) with $a \cdot b > \frac{R(u+vR)}{r}$ also satisfies $ab \equiv u + vr \bmod N$, the analysis for the case $c = a \cdot b + a'$ can be redone by adjusting equation (3) with this probability. This leads to a long formula which is not displayed here but which is discussed in a few examples.

- Assume that the smallest prime factor of N is bigger than r^2. This applies for example to $r = 3$, $N = 2^n - 3$ with $n \not\equiv 3 \bmod 4$. Then $w = 1$ for all u, v as above so that the probability of failure is approximated by $\frac{\phi(N)}{N^2}(\frac{r(r-1)}{2} - \sum_{v=1}^{r-1} v \log \frac{r}{v})$. In the case $r = 3$ this simplifies to $\frac{\phi(N)(3-\log\frac{27}{4})}{N^2}$ which is less than $\frac{1.1}{N}$. Notice that this is much lower than the probability $\frac{3}{N}$ for inputs uniformly distributed in $[0, R^2-1]$.

- Consider the case $r = 38$, $n = 256$ so that $N = 2p$ where p is the prime used in Curve25519 (cf. final part of Section 3.2 on page 13). Then w can only take the values 1 and 2 with

the former occuring if u is odd and the latter if u is even. Thus the probability that $ab \equiv u + vr \bmod N$ is satisfied becomes $\frac{\phi(N)}{N^2} = \frac{p-1}{N^2} \approx \frac{1}{2N}$ if u is odd and $\frac{\phi(N)+2\phi(\frac{N}{2})}{N^2} = \frac{3(p-1)}{N^2} \approx \frac{3}{2N}$ if u is even. Since $u + vR$ can be approximated with vR and the sum over u contains as many even as odd values of u, one obtains the same result as in the analysis of the case $c = a \cdot b + a'$, namely

$$\frac{1}{N}(703 - \sum_{v=1}^{37} v \log \frac{38}{v}) \approx \frac{343}{N}$$

as an approxmiation of the probability of an incorrect result. Again this is lower than in the case of inputs uniformly distributed in $[0, R^2 - 1]$.

4.2 Special arithmetic for Mersenne numbers

In the case of a Mersenne number $N = 2^n - 1$ modular reduction is fairly simple since two reduction steps $\mathcal{R}$ are sufficient and, moreover, the second reduction step consists of adding either 0 or 1. However, if n is not a multiple of the word size, this involves many shifts and in some cases it may be advantageous to interweave these shifts with operations of the preceding multiplication. For example, this is done in [18] which will be outlined in the following.

Let w denote the word size of the underlying architecture and assume that w-bit additions and subtractions, preferably with carry propagation, are available as well as a $w/2 \times w/2$ to w-bit multiplication. Furthermore, assume that the number of words of N is in a range where one can benefit from using the Karatsuba-Ofman technique (or similar ones). Since branches are usually expensive, especially in a SIMD-setting (as in [18]), it is advantageous to adapt the word size in the algorithm so that carries in intermediate results of the Karatsuba-Ofman technique can be avoided. More precisely, let $s < w/2$ be the adapted word size, let $S = 2^s$ be the radix and let m be the number of s-bit words used to represent integers modulo N. Thus up to $w/2 - s$ levels of the Karatsuba-Ofman technique can be applied. If an (unsigned) radix-2^s representation is used, the intermediate coefficients $d_k = \sum_{i,j=0,i+j=k}^{m-1} a_i b_j$ in the product $(\sum_{i=0}^{m-1} a_i S^i) \cdot \sum_{j=0}^{m-1} b_j S^j)$ are bounded by mS^2. This bound can be halved by using a *signed radix-2^s representation*, i.e., writing $A = \sum_{i=0}^{m-1} a_i S^i$ with $-2^{s-1} \leq a_i < 2^{s-1}$ which allows to uniquely represent any integer in the interval $[-2^{ms-1}, 2^{ms-1} - 1]$. Using this representation the intermediate coefficients d_k fit in $\lceil \log_2 m \rceil + 2s - 1$ bits and it is assumed that s and m are chosen such that $\log_2 m + 2s - 1 \leq w$ holds. For such a signed radix-2^s representation the arithmetic operations, i.e., modular addition, subtraction and multiplication as well as conversions between signed radix-2^s and (unsigned) radix-2^w, are described in the following. For $* \in \{+, -, \cdot\}$ the operands of $C \equiv A * B \bmod N$ are written as $A = \sum_{i=0}^{m-1} a_i S^i$, $B = \sum_{i=0}^{m-1} b_i S^i$ and $C = \sum_{i=0}^{m-1} c_i S^i$.

4.2.1 Modular addition and subtraction

These operations are done coefficient wise with a subsequent transformation of the result modulo N into a signed radix-2^s representation. The transformation is facilitated by adding the constant $C_0 = 2^{s-1} \sum_{i=0}^{m-1} S^i$ to A and subtracting it from the result. The steps are as follows:

1. Compute $a'_i = a_i + 2^{s-1}$ for $0 \leq i < m$ (adding C_0).
2. Set $c_i = a'_i + b_i$ in the case of an addition or $c_i = a'_i - b_i$ in the case of a subtraction for $0 \leq i < m$.
3. Set $c = 0$. For $i = 0$ to $m - 1$ first replace c by $c + c_i$, then set c_i to $c \bmod S$ and finally replace c by $\lfloor \frac{c}{S} \rfloor$.
4. Add $c \cdot 2^\alpha$ to c_β where α and β are integers such that $sm - n = \alpha + s\beta$ with $0 \leq \alpha < s$. If $c_\beta \geq S$, propagate the carry as in the previous step until $c = 0$. If handling c_{m-1} produces a carry, repeat this step.

5. Set $c_i = c_i - 2^{s-1}$ for $0 \le i < m$ (subtracting C_0).

In the first step the constant C_0 is added to A, then A and B are added or subtracted in the second step. The third step normalises the coefficients of C to the range $[0, S-1]$ while producing a carry c which corresponds to $c \cdot S^m$ and which is taken care of in the fourth step thus producing a radix-2^s representation of $C + C_0$. In the fifth step C_0 is subtracted so that C is in a signed radix-2^s representation.

4.2.2 Modular multiplication

This is done in two phases. In the first phase the intermediate coefficients $d_k = \sum_{i,j=0,i+j=k}^{m-1} a_i b_j$, $0 \le k < 2m-1$ are computed so that $\sum_{k=0}^{2m-2} d_k S^k$ modulo N is the result C. The second phase handles the reduction of this sum modulo N and its transformation into a signed radix-2^s representation. In the first phase the Karatsuba-Ofman technique (for polynomials) can be used. The reduction of $\sum_{k=0}^{2m-2} d_k S^k$ modulo N proceeds in two substeps. Firstly, the d_k are made non-negative by adding a constant which is 0 modulo N. Next, the d_k with $k \ge m$ are split into two $w/2$-bit words which are added with an appropriate shift to the appropriate $d_{k'}$ with $k' < m$. The resulting sum $\sum_{k=0}^{m-1} d_k S^k$ satisfies $0 \le d_k < 2^w$. The transformation into a signed radix-2^s representation is essentially the same as in the addition method above, i.e., first the constant C_0 is added, next the carries are propagated, and finally the constant C_0 is subtracted. This leads to the following steps:

1. Compute for $0 \le k < 2m-1$ the intermediate coefficients $d_k = \sum_{i,j=0,i+j=k}^{m-1} a_i b_j$ of the product $A \cdot B$ using at most $w/2 - s$ levels of the Karatsuba-Ofman technique.

2. Set $d_0 = d_0 + 2^{w-1}$, $d_i = d_i + 2^{w-1} - 2^{w-1-s}$ for $1 \le i < 2m-1$ and $d_\beta = d_\beta - 2^\alpha$ where α and β are integers such that $s(2m-2) + w - 1 - n = \alpha + s\beta$ with $0 \le \alpha < s$.

3. Set $d_i = d_i + 2^{s-1}$ for $0 \le i < m$ (adding C_0).

4. For $m \le i < 2m-1$ do the following operations. First write $d_i = d_i' + d_i'' \cdot 2^{w/2}$, then add $d_i' \cdot 2^{\alpha'}$ to $d_{\beta'}$ and $d_i'' \cdot 2^{\alpha''}$ to $d_{\beta''}$ where $\alpha', \beta', \gamma', \alpha'', \beta'', \gamma'' \in \mathbb{Z}$ such that $is = \alpha' + s\beta' + n\gamma'$ and $is + w/2 = \alpha'' + s\beta'' + n\gamma''$ with $0 \le \alpha', \alpha'' < s$ and $0 \le \alpha' + s\beta', \alpha'' + s\beta'' < n$.

5. Perform the third and fourth step of the addition method to d_i, $0 \le i < m$.

6. Set $c_i = d_i - 2^{s-1}$ for $0 \le i < m$ (subtracting C_0).

Notice that the m additions of the third step can be combined with corresponding additions in the second step.

4.2.3 Conversions

Usually, the input of a long calculation is available in a radix-2^w representation and this is also the desired format for the result. Therefore, a few conversions between signed radix-2^s and radix-2^w representations are needed at the beginning and at the end. Although in most cases a simple and slow conversion is sufficient, it is possible that during the course of the calculation many conversions are needed so that it must be more efficient. With some of the tricks used above reasonably efficient conversion routines can be built which are sketched below. These can be further optimised by combining some of their steps with steps of the preceding or subsequent modular operation, e.g., the fourth step of the following conversion is the inverse of the first step of modular addition. As above let $A = \sum_{i=0}^{m-1} a_i S^i$ be the signed radix-2^s representation and $\tilde{A} = \sum_{i=0}^{\tilde{m}-1} \tilde{a}_i 2^{wi}$ the radix-2^w representation of the same residue class modulo N with $0 \le \tilde{A} < N$ and an appropriate $\tilde{m}$.

The conversion from radix-2^w to signed radix-2^s proceeds as follows:

1. Add the constant $C_0 = 2^{s-1} \sum_{i=0}^{m-1} S^i$ to $\tilde{A}$ using the standard addition technique with carry propagation.

2. Set $a'_i = 0$ for $0 \leq i < m$. For $0 \leq i \leq \tilde{m}$ split $\tilde{a}_i$ appropriately and add the appropriately shifted parts to the appropriate a'_i.
3. Perform the third and fourth step of the addition method to a'_i, $0 \leq i < m$.
4. Set $a_i = a'_i - 2^{s-1}$ for $0 \leq i < m$ (subtracting C_0).

With the precomputed constant C_1 where $C_1 \equiv -C_0 \bmod N$ and $0 \leq C_1 < N$ the conversion from signed radix-2^s to radix-2^w proceeds as follows:

1. Set $a_i = a_i + 2^{s-1}$ for $0 \leq i < m$ (adding C_0).
2. Initialise the $\tilde{a}_j$ with the precomputed constant C_1.
3. For $0 \leq i \leq m$ split $a_i \cdot 2^{\alpha_i}$ into (at most) two w-bit words where $0 \leq \alpha_i < w$ and $is - n[\frac{is}{n}] \equiv \alpha_i \bmod w$. Then add these words to the appropriate $\tilde{a}_j$ (in order to minimise carry propagation, perform these additions in increasing order of j).
4. Reduce $\sum_{i=0}^{\tilde{m}-1} \tilde{a}_i 2^{wi}$ modulo N; usually this requires only a few shifts and additions.

References

[1] T. Acar. *High-Speed Algorithms & Architectures for Number Theoretic Cryptosystems*. PhD thesis, Department of Electrical & Computer Engineering, Oregon State University, 1997.

[2] T. Acar and D. Shumow. Modular reduction without pre-computation for special moduli. Technical report, Microsoft Research, 2010.

[3] P. Barrett. Implementing the Rivest, Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology (CRYPTO)*, pages 311–323. Springer-Verlag LNCS 263, 1986.

[4] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *Advances in Cryptology (EUROCRYPT)*, pages 92–111. Springer-Verlag LNCS 950, 1994.

[5] K. Bentahar and N. Smart. Efficient 15,360-bit RSA using woop-optimised montgomery arithmetic. In *IMA Cryptography And Coding (IMACC)*, LNCS 4887, pages 346–363. Springer-Verlag, 2007.

[6] D. Bernstein. Error-prone cryptographic designs. Real World Cryptography (RWC) invited talk, 2015.

[7] D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 207–228, New York, NY, USA, Apr. 24–26, 2006. Springer, Heidelberg, Germany.

[8] D. J. Bernstein. Elliptic vs. Hyperelliptic, part I. Talk at ECC (slides at http://cr.yp.to/talks/2006.09.20/slides.pdf), September 2006.

[9] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. In B. Preneel and T. Takagi, editors, *CHES 2011*, volume 6917 of *LNCS*, pages 124–142, Nara, Japan, Sept. 28 – Oct. 1, 2011. Springer, Heidelberg, Germany.

[10] E. Biham. A fast new DES implementation in software. In *Fast Software Encryption (FSE)*, LNCS 1267, pages 260–272. Springer-Verlag, 1997.

[11] D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the American Mathematical Society*, 46(2):203–213, 1999.

[12] J. Bos. *Practical Privacy*. PhD thesis, Technische Universiteit Eindhoven, 1992.

[13] J. Bos, P. Montgomery, D. Shumow, and G. Zaverucha. Montgomery multiplication using vector instructions. In *Selected Areas in Cryptography (SAC)*, LNCS 8282, pages 471–489. Springer-Verlag, 2013.

[14] J. W. Bos. High-performance modular multiplication on the Cell processor. In M. A. Hasan and T. Helleseth, editors, *Workshop on the Arithmetic of Finite Fields – WAIFI 2010*, volume 6087 of *Lecture Notes in Computer Science*, pages 7–24. Springer, 2010.

[15] J. W. Bos, C. Costello, H. Hisil, and K. Lauter. Fast cryptography in genus 2. In T. Johansson and P. Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 194–210, Athens, Greece, May 26–30, 2013. Springer, Heidelberg, Germany.

[16] J. W. Bos, C. Costello, H. Hisil, and K. Lauter. High-performance scalar multiplication using 8-dimensional GLV/GLS decomposition. In G. Bertoni and J.-S. Coron, editors, *CHES 2013*, volume 8086 of *LNCS*, pages 331–348, Santa Barbara, CA, USA, Aug. 20–23, 2013. Springer, Heidelberg, Germany.

[17] J. W. Bos, M. E. Kaihara, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. Solving a 112-bit prime elliptic curve discrete logarithm problem on game consoles using sloppy reduction. *International Journal of Applied Cryptography*, 2(3):212–228, 2012.

[18] J. W. Bos, T. Kleinjung, A. K. Lenstra, and P. L. Montgomery. Efficient SIMD arithmetic modulo a Mersenne number. In E. Antelo, D. Hough, and P. Ienne, editors, *IEEE Symposium on Computer Arithmetic – ARITH-20*, pages 213–221. IEEE Computer Society, 2011.

[19] B. Chevallier-Mames, M. Joye, and P. Paillier. Faster double-size modular multiplication from euclidean multipliers. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 2779, pages 214–227. Springer-Verlag, 2003.

[20] R. E. Crandall. Method and apparatus for public key exchange in a cryptographic system, October 1992. US Patent 5,159,632.

[21] M. Düll, B. Haase, G. Hinterwälder, M. Hutter, C. Paar, A. H. Sánchez, and P. Schwabe. High-speed curve25519 on 8-bit, 16-bit and 32-bit microcontrollers. *Design, Codes and Cryptography*, 77(2), 2015.

[22] N. Ferguson, B. Schneier, and T. Kohno. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley, 2010.

[23] W. Fischer and J.-P. Seifert. Increasing the bitlength of a crypto-coprocessor. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 2523, pages 71–81. Springer-Verlag, 2002.

[24] M. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

[25] P. Gaudry and É. Schost. Genus 2 point counting over prime fields. *J. Symb. Comput.*, 47(4):368–400, 2012.

[26] D. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27:129–146, 1998.

[27] S. Gueron. Enhanced Montgomery multiplication. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 2523, pages 46–56. Springer-Verlag, 2002.

[28] G. Hachez and J.-J. Quisquater. Montgomery exponentiation with no final subtractions: Improved results. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 1965, pages 293–301. Springer-Verlag, 2000.

[29] M. Hamburg. Fast and compact elliptic-curve cryptography. Cryptology ePrint Archive, Report 2012/309, 2012. http://eprint.iacr.org/2012/309.

[30] L. Hars. Long modular multiplication for cryptographic applications. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 3156, pages 45–61. Springer-Verlag, 2004.

[31] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA cryptography specification, version 2.1. Internet Engineering Task Force (IETF) Request for Comments (RFC) 3447, 2003.

[32] M. Joye. On Quisquater's multiplication algorithm. In D. Naccache, editor, *Cryptography and Security: From Theory to Applications*, LNCS 6805, pages 3–7. Springer-Verlag, 2012.

[33] A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. *Physics-Doklady*, 7:595–596, 1963.

[34] M. Knežević, F. Vercauteren, and I. Verbauwhede. Speeding up bipartite modular multiplication. In M. A. Hasan and T. Helleseth, editors, *Arithmetic of Finite Fields – WAIFI*, volume 6087 of *Lecture Notes in Computer Science*, pages 166–179. Springer, 2010.

[35] D. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. 3 edition, 1997.

[36] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[37] A. Lenstra. Generating RSA moduli with a predetermined portion. In *Advances in Cryptology (ASIACRYPT)*, LNCS 1514, pages 1–10. Springer-Verlag, 1998.

[38] A. Lenstra and E. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, 2001.

[39] A. K. Lenstra. Unbelievable security. Matching AES security using public key systems (invited talk). In C. Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 67–86, Gold Coast, Australia, Dec. 9–13, 2001. Springer, Heidelberg, Germany.

[40] A. K. Lenstra and E. R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, Sept. 2001.

[41] H. Lenstra. Factoring integers with elliptic curves. *Annals of Mathematics*, 126(3):649–673, 1987.

[42] V. S. Miller. Use of elliptic curves in cryptography. In H. C. Williams, editor, *CRYPTO'85*, volume 218 of *LNCS*, pages 417–426, Santa Barbara, CA, USA, Aug. 18–22, 1986. Springer, Heidelberg, Germany.

[43] P. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[44] P. Montgomery. Vectorization of the elliptic curve method. Technical report, Centrum Wiskunde & Informatica (CWI), 1994(ish).

[45] G. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8):114–117, 1965.

[46] D. Page and N. Smart. Parallel cryptographic arithmetic using a redundant montgomery representation. *IEEE Transactions on Computers*, 53(11):1474–1482, 2004.

[47] P. Pailler. Low-cost double size modular exponentiation or how to stretch your cryptocoprocessor. In *Public Key Cryptography (PKC)*, LNCS 1560, pages 223–234. Springer-Verlag, 1999.

[48] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 1 edition, 2000.

[49] J.-J. Quisquater. Encoding system according to the so-called RSA method, by means of a microcontroller and arrangement implementing this system. U.S. Patent 5,166,979.

[50] R. Rivest. A description of a single-chip implementation of the RSA cipher. *LAMBDA Mazazine*, 1(3):14–18, 1980.

[51] R. Rivest, A. Shamir, and L. Adleman. Cryptographic communications system and method. U.S. Patent 4,405,829.

[52] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM (CACM)*, 21(2):120–126, 1978.

[53] A. Sherman. *VLSI Placement and Routing: The PI Project*. Springer-Verlag, 1989.

[54] S. Singh. *The Code Book: The Secret History of Codes and Code-Breaking*. Fourth Estate, 1999.

[55] N. Smart. Algorithms, key size and parameters report. Technical report, European Union Agency for Network and Information Security (ENISA), 2014. http://www.enisa.europa.eu.

[56] J. A. Solinas. Generalized Mersenne numbers. Technical Report CORR 99-39, Centre for Applied Cryptographic Research, 1999. http://www.cacr.math.uwaterloo.ca/techreports/1999/corr99-39.pdf.

[57] U.S. Department of Commerce/National Institute of Standards and Technology. Digital Signature Standard (DSS). FIPS-186-4, 2013. http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

[58] S. Vanstone and R. Zuccherato. Short RSA keys and their generation. *Journal of Cryptology*, 8(2):101–114, 1995.

[59] C. Walter. Faster modular multiplication by operand scaling. In *Advances in Cryptology (CRYPTO)*, LNCS 576, pages 313–323. Springer-Verlag, 1991.

[60] C. Walter. Montgomery exponentiation needs no final subtractions. *Electronics Letters*, 35(21):1831–1832, 1999.

[61] C. Walter. Montgomery's multiplication technique: How to make it smaller and faster. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 1717, pages 80–93. Springer-Verlag, 1999.

[62] M. Wiener. Cryptanalysis of short RSA secret exponents. *IEEE Transactions on Information*, 36(3):553–558, 1990.

[63] M. Yoshino, K. Okeya, and C. Vuillaume. Double-size bipartite modular multiplication. In *Australasian Conference on Information Security and Privacy (ACISP)*, LNCS 4586, pages 230–244. Springer-Verlag, 2007.