# Elliptic Curve Cryptography for the masses: Simple and fast finite field arithmetic

May 6, 2024

**Abstract**

Shaped prime moduli are often considered for use in elliptic curve and isogeny-based cryptography to allow for faster modular reduction. Here we focus on the most common choices for shaped primes that have been suggested, that is pseudo Mersenne, generalized Mersenne and Montgomery friendly primes. We consider how best to to exploit these shapes for maximum efficiency, and provide an open source tool to automatically generate, test and time working high-level language finite-field code. Next we consider the advantage to be gained from implementations that are written in assembly language and which exploit special instructions, SIMD hardware if present, and the particularities of the algorithm being implemented.

## 1 Introduction

In the early days of discrete log based cryptography over the finite field $\mathbb{F}_p$ an obvious idea was to use a prime $p$ of a shape that allowed faster modular reduction. However there was a concern that any useful special shape also significantly weakened the discrete log problem, on which security depended. The problem was that this discrete logarithm problem is subject to an "index calculus" attack. And a useful shaped prime may allow a much faster index calculus attack [21]. As bluntly stated in [19] "primes of a special form allow easier computation of discrete logarithms".

But this changed with the discovery of elliptic curve cryptography, as in the case of an elliptic curve defined over a finite field there is no index calculus attack (loosely speaking because while integers can be factored, points on curves cannot). So a shaped modulus is perfectly acceptable, and indeed widely used.

It is generally agreed that a Mersenne prime would be best for modular reduction in this context – but with the exception of $2^{127}-1$ and $2^{521}-1$ they are rarely of an ideal size. So the next best idea would appear to be a pseudo-Mersenne prime $2^m - c$ for some small $c > 1$. And this is how things

would probably have progressed when it came to standardising elliptic curve cryptography. Then a spanner was thrown into the works [3], [5].

Richard Crandall was a prominent mathematician who, quickly realising the potential of pseudo-Mersene primes, went ahead and patented them. Which at the time generated a very negative reaction. The idea of patenting numbers of a particular form seemed indefensible. In any case the patent was never enforced, and is now completely irrelevant.

Nonetheless the patent application appeared at around the same time as NIST was considering standardizing elliptic curves. And it had sufficient chilling impact that NIST decided to look elsewhere for their primes. So they settled instead on the definitely-not-patented generalised Mersenne primes (sometimes referred to as Solinas primes, after NIST's Jerome Solinas) of the form $2^m - 2^a \pm 2^b \pm 2^c ... \pm 1$, ideally with a minimal number of intermediate terms. For optimal performance $a, b, c..$ were chosen to be multiples of 32, as in those days 32-bit architectures were dominant.

Once it was realised that Crandall's patent was a dud, pseudo-Mersenne primes were again preferred, famously in Bernstein's curve25519 proposal [6], itself subsequently standardized [29]. That might have been considered the end of the matter, until Hamburg suggested the Goldilocks curve [25], also standardised in [29], and based on the generalised Mersenne prime $2^{448} - 2^{224} - 1$. At around the same time an honourable effort was made to introduce alternate improved standards [11], but these efforts did not achieved much traction.

Which leaves the question still open – which shape is best? However given that curve standards are by now well established, any further debate is probably moot. We will have to work with the standards we have. But having said that, if isogeny based cryptography ever becomes viable, there will be a need in that context for some new standard curves, in which case it might be appropriate to look again at the question of the ideal prime shape.

The latest version 1.3 of the well known TLS protocol supports the use of the NIST256 (aka secp256r1, generalised Mersenne), NIST384 (aka secp384r1, generalised Mersenne), NIST521 (aka secp521r1, Mersenne), C25519 (pseudo-Mersenne) and C448 (generalised Mersenne) elliptic curve fields for key exchange and signature (see sections 4.2.3 and 4.2.7 of [56]).

In this study we will consider high level language implementations simple enough to be generated from a Python script. Then we will investigate the speed-up that can be expected from an optimal asssembly language implementation.

## 2 Comparing prime shapes

- Pseudo Mersennes (PMs) of the form $2^m - c$ are plentiful, it is not hard to find one of any size, and given the density of primes it is always

possible to find one where $c$ fits comfortably in a single computer word. There are so many in fact that other criteria can be applied which may lead to slight improvements (for example $p = 3 \bmod 4$ may be preferred as it allows faster modular square roots). But all other things being equal, $c$ should be as small as possible. Pseudo Mersennes are also largely indifferent to word size, and will work equally well on 32-bit and 64-bit architectures. For the pseudo Mersennes essentially the same optimal modular reduction algorithm can be deployed in every case, as $c$ is just a one-word parameter. It would be straightforward to write a general purpose library to efficiently implement a variety of elliptic curves over a range of pseudo-Mersenne primes.

- Generalised Mersennes (GMs) of the form $2^m - 2^a \pm 2^b \pm 2^c ... \pm 1$ on the other hand are much harder to find, and hence multiple terms $a, b, c..$ may be required to find a prime, and this leads to inefficiency. This is especially true if all of $a, b, c..$ are required to be multiples of the radix. We would obviously prefer the minimal number of terms. Generalised Mersennes are usually bound to the choice of radix which depends on the word length. The standardised NIST curves are mostly only a good fit for a 32-bit architecture, as while all of $a, b, c..$ may be divisible by 32, some of them are not divisible by 64. A specialised routine must be written for every individual generalised Mersenne in order to ellicit the best performance.

- Montgomery-friendly primes (MFs) are of the form $k.2^m - 1$ [3] [11], and as the name implies they are particularly efficient if using Montgomery reduction. Unfortunately their arrival on the scene was rather too late for them to be considered for elliptic curve standardization. But they have found a new relevance in the context of isogeny-based cryptography [4], [48]. Such primes are particularly fortunate when $k$ is small enough to fit in a single limb.

On the face of it pseudo-Mersennes appear to win over generalised Mersennes on every count, and we don't doubt that but for Crandall's intervention generalised Mersennes may never have been considered. Nonetheless three of the five curves supported by TLS1.3 use generalised Mersennes (and one NIST521, which is actually a true Mersenne, can be considered to be member of either camp – leaving C25519 as the only true pseudo-Mersenne).

# 3 Implementation choices

First we will outline the constraints that apply to this study. We do not consider an implementation where the prime moduli is specified at run time, rather than at compile time, as is done in many popular general purpose

cryptographic libraries like RELIC [2]. This approach does not permit full exploitation of the prime shape, and therefore will always be at a disadvantage.

We also do not consider larger prime moduli which are unshaped (or "dense") as used in the well known RSA method. And although the moduli that arise in the context of pairing-based cryptography [40] do often have a shape, its a shape which apparently cannot be exploited for faster performance. So we do not consider those primes.

We are primarily interested in modular arithmetic involving numbers in the range 0 to $p-1$, where the shaped prime modulus $p$ will be of a fixed size of $m$ bits typically in the range 256–512. And we will focus on the more time consuming primitive operations, modular multiplication and squaring.

Assume a computer of word length $w$ bits, where $w = 32$ or $w = 64$. Then these big numbers will be represented with $n$ words, or *limbs*, using a radix $2^b \leq 2^w$. Clearly a radix that is a power of 2 will be optimal on a binary computer.

Modular multiplication/squaring consists of two parts, integer multiplication/squaring followed by modular reduction, modulo the prime $p$. Integer multiplication for numbers of these sizes is essentially an $O(n^2)$ process in terms of the number of limbs $n$. In fact using school-boy methods exactly $n^2$ partial products must be accumulated, and if using one level of classic Karatsuba this reduces to $3n^2/4$ partial products. By using a shaped prime, the hope is that reduction will be $O(n)$, and hence comparatively negligible.

The first decision is which radix to use. Two approaches are now possible, the first being to use a *saturated* representation where the radix is equal to the wordlength, or *unsaturated* where the radix is somewhat less than the wordlength. This design decision is surprisingly impactful and has multiple implications. Using a saturated representation holds out the promise of a representation with the minimal number of limbs. This can be quite important, particularly if the modulus size is an exact multiple of the word length, as is commonly the case. Using a radix of $2^{64}$ on a 64-bit computer for a 256-bit modulus requires just 4 limbs. Using an unsaturated radix will require 5 limbs, and $5^2$ is of course significantly (36%) more than $4^2$.

However if using a saturated radix then the issue arises of handling a carry bit, generated by the addition of limbs, as high level languages have no direct access to a carry bit. A carry bit can of course be simulated, but in our view this introduces a slow down which makes a high-level language saturated-radix implementation unattractive. With an unsaturated radix carry propagation becomes a simple matter of shifting and masking, easily and cheaply implemented in a high level language. Moreover propagation can be delayed, as carries can exceed a single bit.

Therefore our scripts will only use an unsaturated radix. At this point it would appear that the prospects for this approach are not promising in the context of some well established curves. Indeed in [16] the authors

say "unsaturated-arithmetic performance degrades rapidly when used with moduli that it is not a good match for, so older curves such as P-256 need to be implemented using saturated arithmetic".

Note that in the interests of keeping our scripts as simple as possible, we do not exploit the clever mixed-radix idea first suggested by Bernstein [6]. For example for the $2^{255} - 19$ modulus on a 32-bit processor, it is beneficial to use an alternating radix for each limb of $2^{26}, 2^{25}, 2^{26}, 2^{25}$.. which provides a better fit.

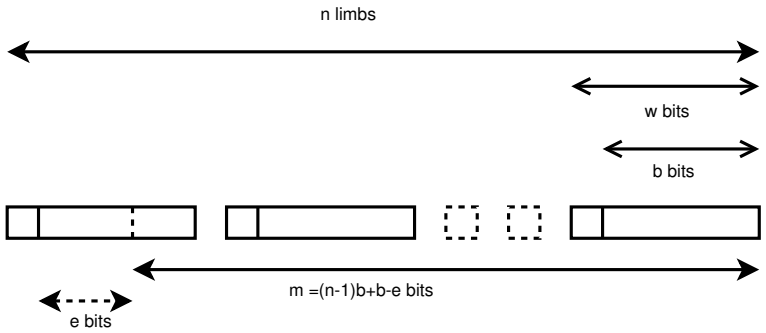


Figure 1: Unsaturated radix representation of $m$-bit Big number

Methods for integer multiplication and squaring involve the accumulation of double-precision partial products. Therefore a high level language (HLL) should ideally have a capability to handle the double precision product of two word-sizes integers. Unfortunately while most 32-bit oriented HLL compilers support a 64-bit double precision type, most 64-bit oriented HLL compilers do not support a 128-bit type. There is no good reason for this anomaly. However there are two important exceptions, Rust (as part of its standard) and C/C++ (unofficially). Fortunately these are the most likely languages to be used in a performant HLL implementation of elliptic curve cryptography. So we will assume that a 128-bit integer type is available for use on 64 bit processors.

For our script-derived implementations we will consider two methods for integer multiplication and two methods for modular reduction. For multiplication we will either use the school-boy method using product scanning, or the ADK Karatsuba algorithm [50]. We prefer this Karatsuba variant method as its implementation is independent of the number of limbs in the representation, and hence easily generated from a script. For modular reduction we consider one method based on Montgomery arithmetic [39] for primes of any (or no) shape, and another based on what might be called the folklore method as it applies to pseudo-Mersenne moduli. See [36] algorithm 14.47.

### 3.1 Choosing the unsaturated radix

Choosing the optimal radix can be quite challenging, particularly in the context of a 64-bit processor where multiple choices all allow the same minimal number of limbs. There may be other constraints. For example it may be preferred that there is support for lazy reduction, so the value of $e$ (figure 1) may be desired to be non-zero. Other considerations specific to the prime shape may also impact the choice of radix. In the context of a generalised Mersenne it might be preferred that the radix exactly divide some or all of the powers of 2 that constitute the prime. Clearly an implementation should be flexible in terms of the radix choice.

For example consider the choice of radix for the NIST521 curve. If considered as a pseudo-Mersenne a nice choice would be 58 bits [22]. However if considered as a generalised Mersenne and using the Montgomery reduction method this radix only allows an excess of 1 bit. In the context of elliptic curve cryptography, we would like an excess of at least 2 bits (see below). Therefore a radix of 60 bits might be preferrable here.

On a 32-bit processor there is much less flexibility, and not all of our preferences may be realisable, consistent with keeping the number of limbs to a minimum.

Of course in all cases extreme care must be taken to avoid the possibility of overflow (see below). Note that random testing is not sufficient, as failure cases can be extremely rare.

### 3.2 Hardware

This study makes use of a range of computing platforms, which we list and nickname here

- An 8th generation i5-8250U Intel x86-64 processor (Skylake)
- A 13th generation i7-1360P Intel x86-64 processor (Raptorlake)
- An Apple Mac Mini with an Apple M2 processor (Apple M2)
- A Raspberry Pi 5, with a 64-bit Cortex-A76 processor (RP5)
- A LicheePi 4A (RISC-V) 64-bit processor (LicheePi)
- An STM32F407G-DISC development board, with a 32-bit ARM Cortex-M4 processor (STM32F4)
- An Arduino Nano RP2040 with an ARM Cortex-M0 processor (RP2040)
- A TinyPICO board powered by a 32-bit ESP32 processor, clocked at 240MHz (TinyPICO)

- A Teensy 3.2 board powered by a 32-bit ARM Cortex-M4 processor, clocked at 48MHz (Teensy)

On some of these platforms performance is measured in clock cycles (using Bernsteins libcpucycles library [7] if available), in other cases it is measured in nanoseconds or microseconds.

Where head-to-head comparisons are made, they are conducted on the same physical device whenever possible to ensure fairness

### 3.3 Compilers

For each architecture we had up to 3 compilers to choose from. On the Intel processors we had access to gcc version 11.4, clang version 14.0, and Intel's own compiler icx version 2024.0.2. On the Raspberry PI we had the choice of gcc version 13.2 and clang version 16.06. On the LicheePi 4A RISC-V we again used gcc version 13.2. For the Apple mac we use clang version 15 (disguised as gcc), and gcc version 13.2.

Naturally we prefer the compiler which for each modulus produces the fastest code. No individual compiler was best in all cases, each was optimal for at least one modulus. In some cases different compilers were better for modular multiplication and for modular squaring, and in theory outputs from both could be linked together. But we did not pursue this possibility.

### 3.4 Moduli

Unfortunately there is no agreed convention for the naming of prime moduli. Since there are a massive number of possibilities we restrict to a popular subset, sufficient to draw our conclusions. Our naming scheme aims for brevity rather than consistency.

- C25519 is the pseudo-Mersenne prime made famous by Daniel Bernsten, $2^{255} - 19$
- NIST256 is the NIST 256-bit standard generalised Mersenne prime also known as p256 and secp256r1
- NIST384 is the NIST 384-bit standard generalised Mersenne prime also known as p384 and secp384r1
- C448 is the Goldilocks generalised Mersenne prime $2^{448} - 2^{224} - 1$
- NIST521 is the NIST 521-bit standard Mersenne prime also known as p521 and secp521r1, $2^{521} - 1$
- SIDH434 is the SIDH Montgomery-friendly prime $2^{216}.3^{137} - 1$
- SIDH751 is the SIDH Montgomery-friendly prime $2^{372}.3^{239} - 1$

- MFP4 is the isogeny-useful Montgomery-friendly prime $3.67.2^{246} - 1$ [48]
- MFP4096 is the large 4096-bit isogeny-useful Montgomery-friendly prime from [13]
- PM266 is the pseudo-Mersenne prime $2^{266} - 3$
- PM512 is the pseudo-Mersenne prime $2^{512} - 569$
- PM336 is the pseudo-Mersenne prime $2^{336} - 3$
- PM383 is the pseudo-Mersenne prime $2^{384} - 183$
- C414 is the pseudo-Mersenne prime $2^{414} - 17$
- GM266 is the generalised-Mersenne prime $2^{266} - 2^{168} - 1$

### 3.5 Online resources

Many researchers have gone to the effort to make their code publicly accessible and easy to benchmark, and for this we are grateful. These are the ones that we used most often, but needlesss to say the list is not exhaustive, and we may well have missed some important contributions.

- Nath and Sarkar have provided assembly language implementations for many pseudo-Mersenne primes and also the generalised-Mersenne modulus C448. They also provide code for an AVX2 SIMD supported implementation of RFC7748. See [43], [42] and [41].
- The Amazon Web Services saturated radix assembly language code for 64 and ARM64 [27]
- An early C implementation of C25519 from Adam Langley [28]
- Code from Thomas Pornin's implementation of TLS [44], his crrl rust library [46], and also [45]
- The fast assembly language implementations by Emil Lenngren, for example [32], also [30] for a NEON SIMD supported implementation of RFC7748.
- The C and assembly language code provided by Microsoft researchers for implementation of the doomed SIKE isogeny-based protocol [38]
- The example outputs of code generated from the Fiat-crypto project [16] for a range of moduli
- The RISC-V assembly language code from [14]
- The CSIDH code associated with the paper [13]

# 4 Avoiding overflow

One issue that needs to be addressed is the avoidance of overflow, and this in turn may impact on our choice of radix. There are two types of overflow that may occur. Limb overflow occurs where a computed limb value exceeds $2^w$, or accumulated double precision partial products exceed $2^{2w}$. Modulus overflow occurs where a computed multiprecision number exceeds the modulus $p$.

## 4.1 Limb overflow

Consider $0 \leq x, y < p$ and the product $M = x.y$, where $x$ and $y$ are represented by $n$ limbs using a radix $2^b$. The digits of $x$ are the positive values $x_0, x_1..x_{n-1}$ where $x_i < 2^b$, similarly $y$, and the digits of the product $M$ are $M_0, M_1..M_{2n-2}$, where each digit $M_i$ is a double precision value.

Since the $M_i$ may be the sum of up to $n$ double precision partial products, this constrains the value of $b$ as we must avoid the possibility of overflowing the double precision type, therefore $n.(2^b - 1)^2 < 2^{2w}$. This constraint is combined with the desire to make $n$ as small as possible. And in case of a tie-break, by default we choose the smallest value of $b$. This strategy usually dictates a good choice for $b$. But keeping $n$ as small as possible should be the overriding concern.

Limb overflow can also arise as a result of failing to propagate carries after a modular addition or subtraction. In fact this is quite a common optimization [16]. However we regard it as dangerous, as overflow may occur due to function misuse, and the use of non-reduced limbs can have a secondary impact on the above assumptions. Therefore the outputs of our functions will (almost) always be strictly represented with carries fully propagated.

## 4.2 Modulus overflow

It is surprisingly difficult to guarantee that the output of a modular reduction is strictly less than $p$. But it is surprisingly easy to guarantee that it is less than $2p$. This applies to both Montgomery reduction and pseudo-Mersenne reduction. To achieve full reduction would require a conditional subtraction of the modulus after every operation. Therefore we will loosen this constraint and allow the outputs of our functions to be only less than $2p$. But this in turn means that these functions should tolerate inputs that are not fully reduced. For pseudo-Mersennes this is not an issue, but when using Montgomery reduction the guarantee that the output be less than $2p$ depends on a constraint on the inputs. The solution (Bos and Montgomery [10]) is to ensure an excess $e$ of at least 2 bits. Now we can be assured that a sequence of modular multiplications and squarings without a final

conditional subtraction cannot result in overflow.

However elliptic curve point addition and doubling formulae interleave modular multiplications with modular additions and subtractions. Therefore to avoid overflow, given inputs less than $2p$, the outputs of these functions must also be less than $2p$. This would seem to require a conditional subtraction. However depending on the context in which the finite field arithmetic is used, this may be avoided using the technique of lazy reduction as described in [51]. Assuming that the higher level code implements a "stable" calculation in the sense of [51], then modular addition can proceed with carry propagation, but without modular reduction. Modular subtraction can be executed by adding a multiple of the modulus to the result, and again proceeding with carry propagation without modular reduction. The modulus multiple to use can be determined by a simple analysis as described in [51]. In the case of RFC7748 the appropriate multiple is 2.

At the end of a calculation a conditional subtraction of the modulus $p$ will be required to get the unique result modulo $p$ in the required range, prior to serialization and output.

# 5 Lucky primes

We only need to be lucky and find one good prime at each desired security level. Consider for example the Mersenne prime $2^{521} - 1$. In [22] the fact that 522 is an exact multiple of 29, and also of 58 make $2^{29}$ and $2^{58}$ very lucky choices for an unsaturated radix implementation on both 32-bit and 64-bit architectures respectively. For the famous prime $2^{255} - 19$, observe that 255 is a multiple of 51, which makes a 5-limb unsaturated radix a nice choice for a 64-bit processor. In the context of a generalised Mersenne it will be preferred that the radix exactly divide some or all of the powers of 2 that constitute the prime. The Goldilocks prime [25] $2^{448} - 2^{224} - 1$, despite being a generalised Mersenne, is particularly fortuitous as 448 and 224 are both multiples of 28 and 56. Indeed this prime was chosen in part for its ease of implementation using an unsaturated radix on both 32-bit and 64-bit processors. Another example of a "lucky" prime would be $2^{336} - 3$ [49] as $c = 3$ is small and again 336 is divisible by 28 and 56.

With the benefit of hindsight there is little doubt that doubt that standardized curves should have used more suitable moduli. And so we could not avoid the temptation of introducing the nice GM prime $2^{266} - 2^{168} - 1$ in this study.

# 6 The scripts

Rather than tediously generate code on a case-by-case basis, Python scripts were developed which automatically generate simple yet efficient high level

language code. These Python scripts are available for download [1]. Since Python natively supports multi-precision arithmetic, the generated code is also tested internally with random inputs and its output checked for correctness (although this by itself would not be sufficient. So further checks against the possibility of overflow are also carried out). Where possible a program is also created which when executed reports timings for modular multiplication, squaring and inversion, both in clock cycles and nano-seconds.

Our scripts generate a complete set of finite field functions, suitable for an implementation of cryptography based on elliptic curves. We are solely interested in the field over which an elliptic curve group is defined and so we are indifferent to the type of curve (Weierstrass, Edwards, Montgomery) or its purpose (key exchange, signature, encryption).

We have developed two sets of scripts, one that generates code in C and another in Rust. Here we consider only the C version.

Note that our script-derived code represents one extreme end of a spectrum where the code generation process is indifferent to the processor architecture, the elliptic curve and the protocol being implemented. At the other end of the spectrum might be an arduous hand-written assembly language implementation which is targeted at a specific architecture, and a specific elliptic curve for use in the context of a specific protocol. The hoped for advantage of the latter approach is that it should be a lot faster (but by how much?).

For a complete arithmetic we will need to generate code for modular addition, subtraction and multiplication/squaring. But also important is code for modular inversion and square roots. Here we will use a method based on Fermat's little theorem [52] using a near-optimal addition chain provided courtesy of Michael McLoughlin's remarkable tool [34], which is invoked automatically from inside of our Python scripts.

All of the functions generated are written to execute in constant time. However we take no responsibility for any non-constant time behaviour introduced by the compiler (although we make some effort to discourage this), nor for non-constant time instruction execution by the processor hardware.

A useful benchmark is an implementation of the RFC7748 key exchange algorithm [29]. Uniquely this algorithm can be built directly on top of a finite field implementation, as it uses an $x$-only representation of elliptic curve points on Montgomery curves. The algorithm requires a mix of modular multiplications, squarings, additions and subtractions which is typical of the mix of operations required for elliptic curve cryptography. So we also provide a framework into which our finite field code can be dropped, which then compiles to implement RFC7748.

[1] `https://github.com/mcarrickscott/modarith`

# 7 Modular reduction – pseudo Mersennes

The basic folklore algorithm for modular reduction modulo a pseudo-Mersenne prime exploits the fact that $2^m = c \bmod p$. After carries are propagated the top $m$ bits of the product should be detached, multiplied by $c$, the result added to the lower $m$ bits and carries propagated. The first problem with this is that the split between the top and bottom $m$ bits probably falls mid-limb. (But not for a lucky prime like C25519 on a 64-bit processsor using a base of $2^{51}$, where the split falls nicely between the bottom and top 5 limbs). However there is a simple fix for this. Note that the modular reduction is not complete, as the top $m$ bits multiplied by $c$ will be greater than $m$ bits. So we will not be finished after this stage. Therefore perform the reduction instead modulo $2^{m+e} - c.2^e$, where $e$ is the "excess" shown in figure 1. Now the split does fall between limbs. Note that in [22] the same idea is used, and the first reduction is actually performed modulo $2^{522} - 2$.

However as noted above the reduction is not complete. So next extract the bits from the $m$-th bit position upwards, multiply this number by $c_p = c.2^e$ and add the result to the lower $m$ bits, propagating the carries. Now we have a result less than $2p$ assuming $c_p < 2^b$ and $n > 2$ which will always be true for cases of interest.

But there is a better way. The multiplication and the first part of the reduction can be merged [8], [22], [43]. The basic idea is for each row of the multiprecision multiplication to sum the relevant partial products, and then continue to accumulate the partial products that will be multiplied by $c_p$, and that apply to the same row. Only when the two contributions are combined are the carries finally propagated. For example the 4th digit of the partially reduced product of two 9-limbed numbers will then be calculated as

$$M_3 = x_0y_3 + x_1y_2 + x_2y_1 + x_3y_0 + c_p(x_4y_8 + x_5y_7 + x_6y_6 + x_7y_5 + x_8y_4)$$

This strategy works best with a smaller $c$ and a smaller $e$, as otherwise it introduces a new threat of overflow. But if the multiplication of the second set of partial products by $c_p$ could cause an overflow, then Nath and Sarkar [43] suggest a simple "fix". Reduce the second accumulation of partial products into two $b$-bit digits, and add the upper part to the next digit of $M$, combining it with the lower part that arises for that next digit. Ideally if $c$ is small enough this extra step should not be necessary, but note that this fix is much more likely to be required on a 32-bit processor.

In the final stage of the reduction we follow Nath and Sarkar and do not propagate the carries beyond the second least significant limb, as except in rare cases carry propagation quickly peters out. So we will allow the second

least significant limb to lie in the range $0..2b$ and take this into account with our overflow analysis.

# 8 Modular reduction – generalised Mersennes

Here the literature is a little confusing. The "text-book" method is an adaption of the folklore method to the generalised prime shape. See section 2.2.6 of [26]. However as pointed out by Guneysu and Paar [24] multiple correction steps might be required to get the final result into the desired range. In a software implementation this could be very expensive. So in practise implementors have turned instead to using the Montgomery method for modular reduction [39], in part because with a single pass the result is guaranteed to fall into the range $0..2p$. In particular see the influential paper by Gueron and Krasnov [23], whose implementation was offered as a patch for OpenSSL, and the fast code by Lenngren [32]. For an understanding of Montgomery arithmetic, see [39], [3], [36]. See also [9].

At first glance it may appear that given the unsuitability of an unsaturated representation for the NIST primes, that nothing better than a generic Montgomery reduction is possible. But this is not the case. In fact a generalised Mersenne maintains much exploitable shape even without using the ideal saturated radix [23]. And notably the C448 modulus [25] was designed specifically to keep its shape even in the unsaturated setting.

## 8.1 Non-shaped primes

We start with an algorithm for modular reduction on a 64-bit processor, modulo the completely unshaped Brainpool 256-bit modulus [33], $p$ = `0xA9FB57DBA1EEA9BC3E660A909D838D726E3BF623D52620282013481D1F6E5377`. See algorithm 1. Montgomery reduction [39] requires the precomputed constant $n' = 1/(R - p) \bmod q$, where $q$ is the unsaturated radix $2^{52}$ and $R$ is the smallest power of two which is a multiple of the wordlength, and is greater than $p$. Also required is the constant $c = R^2 \bmod p$.

Observe that here we are using a simple product-scanning method to generate the accumulation of the double precision partial products that arise from multiplying $x$ by $y$. Clearly in the case of modular squaring we will do better, as the same partial products repeat. Alternatively Karatsuba-like methods could also be deployed for both multiplication and reduction, see for example [50]. However for the purposes of the current exposition we will stick to simple product-scanning.

An element $x \in \mathbb{F}_p$ can be converted to Montgomery form as $x \leftarrow modmul(x, c)$. It can be converted back to normal form as $x \leftarrow modmul(x, 1)$. Recall that this conversion to/from the Montgomery domain is a once-off calculation applied to inputs, outputs and constants, and

the main computation in an elliptic curve context is carried out entirely in the Montgomery domain.

To avoid overflow it is important that the double precision accumulation in the variable $\mathbf{t}$ does not overflow $2^{128}$. Clearly if overflow does not occur on line 13 of algorithm 1, then it will not be a problem elsewhere, but this must be checked making suitable worst-case assumptions.

**Algorithm 1** Modular multiplication with Brainpool 256-bit modulus, 64-bit processor

INPUT: Inputs $x, y < 2p$, radix is $2^{52}$
OUTPUT: $z = x.y \bmod 2p$

1: **function** MODMUL$(x, y)$
2: $p_0 =$ `0x3481d1f6e5377`
3: $p_1 =$ `0x23d5262028201`
4: $p_2 =$ `0xd838d726e3bf6`
5: $p_3 =$ `0xa9bc3e660a909`
6: $p_4 =$ `0xa9fb57dba1ee`
7: $q = 2^{52}$
8: $n' =$ `0x75590cefd89b9`
9: $\mathbf{t} \leftarrow x_0 y_0 \quad v_0 \leftarrow (\mathbf{t}n') \bmod q \quad \mathbf{t} \leftarrow \mathbf{t} + v_0 p_0 \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
10: $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=0}^{1} x_i y_{1-i} + v_0 p_1 \quad v_1 \leftarrow (\mathbf{t}n') \bmod q \quad \mathbf{t} \leftarrow \mathbf{t} + v_1 p_0 \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
11: $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=0}^{2} x_i y_{2-i} + v_0 p_2 + v_1 p_1 \quad v_2 \leftarrow (\mathbf{t}n') \bmod q \quad \mathbf{t} \leftarrow \mathbf{t} + v_2 p_0 \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
12: $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=0}^{3} x_i y_{3-i} + v_0 p_3 + v_1 p_2 + v_2 p_1 \quad v_3 \leftarrow (\mathbf{t}n') \bmod q \quad \mathbf{t} \leftarrow \mathbf{t} + v_3 p_0 \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
13: $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=0}^{4} x_i y_{4-i} + v_0 p_4 + v_1 p_3 + v_2 p_2 + v_3 p_1 \; v_4 \leftarrow (\mathbf{t}n') \bmod q \; \mathbf{t} \leftarrow \mathbf{t} + v_4 p_0 \; \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
14: $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=1}^{4} x_i y_{5-i} + v_1 p_4 + v_2 p_3 + v_3 p_2 + v_4 p_1 \quad z_0 \leftarrow t \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
15: $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=2}^{4} x_i y_{6-i} + v_2 p_4 + v_3 p_3 + v_4 p_2 \quad z_1 \leftarrow t \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
16: $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=3}^{4} x_i y_{7-i} + v_3 p_4 + v_4 p_3 \quad z_2 \leftarrow t \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
17: $\mathbf{t} \leftarrow \mathbf{t} + x_4 y_4 + v_4 p_4 \quad z_3 \leftarrow t \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
18: $z_4 \leftarrow t$
19: **return** $z$
20: **end function**

## 8.2 NIST256

Consider next the NIST prime

$$p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$$

Using a radix of $2^{52}$ this can be represented in five limbs as {`0xfffffffffffff`, `0xfffffffffff`, `0x00`, `0x1000000000`, `0xffffffff0000`}, and we can already see some scope for optimization. But importantly this prime is "Montgomery friendly" [3], because $n' = 1$. Also $p_2 = 0$. By allowing negative digits, the representation can be considered as {$-$`0x1`, `0x100000000000`, `0x0`, `0x1000000000`, `0xffffffff0000`}. See [23], observation 3. Observe also that some prime digits are now exact powers of 2, and therefore some multiplications can be replaced by shifts. Applying all these optimizations, we arrive at algorithm 2, a tailored modular multiplication algorithm for the NIST256 prime. See algorithm 2.

**Algorithm 2** Modular multiplication with NIST 256-bit modulus, 64-bit processor

INPUT: Inputs $x, y < 2p$, radix is $2^{52}$
OUTPUT: $z = x.y \bmod 2p$

**function** MODMUL$(x, y)$
  $p_4 =$ `0xfffffffff0000`
  $q = 2^{52}$
  $\mathbf{t} \leftarrow x_0 y_0 \quad v_0 \leftarrow \mathbf{t} \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
  $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=0}^{1} x_i y_{1-i} + v_0 2^{44} \quad v_1 \leftarrow \mathbf{t} \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
  $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=0}^{2} x_i y_{2-i} + v_1 2^{44} \quad v_2 \leftarrow \mathbf{t} \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
  $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=0}^{3} x_i y_{3-i} + (v_0 + v_2 2^8) 2^{36} \quad v_3 \leftarrow \mathbf{t} \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
  $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=0}^{4} x_i y_{4-i} + v_0 p_4 + (v_1 + v_3 2^8) 2^{36} \; v_4 \leftarrow \mathbf{t} \bmod q \; \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
  $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=1}^{4} x_i y_{5-i} + v_1 p_4 + (v_2 + v_4 2^8) 2^{36} \quad z_0 \leftarrow t \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
  $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=2}^{4} x_i y_{6-i} + v_2 p_4 + v_3 2^{36} \quad z_1 \leftarrow t \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
  $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=3}^{4} x_i y_{7-i} + v_3 p_4 + v_4 2^{36} \quad z_2 \leftarrow t \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
  $\mathbf{t} \leftarrow \mathbf{t} + x_4 y_4 + v_4 p4 \quad z_3 \leftarrow t \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
  $z_4 \leftarrow t$
  **return** $z$
**end function**

Clearly by inspection this will be superior to the generic method as applied to the Brainpool modulus.

### 8.3 C448

The approach described above works particularly well for the C448 prime, designed as it was to be "unsaturated-friendly", as well as being Montgomery-friendly. The prime is

$$p = 2^{448} - 2^{224} - 1$$

And using a radix of $2^{56}$, again allowing negative digits, it can obviously be described as $\{-$`0x1`$, $`0x0`$, $`0x0`$, $`0x0`$, -$`0x1`$, $`0x0`$, $`0x0`$, $`0x0`$, $`0x1`$\}$. See algorithm 3

An issue that arises in this case is the possibility of negative digits appearing in the output. To prevent this the full radix is added to the calculation of each affected digit, and compensated for by subtracting 1 from the next higher digit. Hence the appearance of the term $q - 1$ in the algorithm. Observe that very little computation is required over and above that which would be required for multiplication without reduction.

## 9 Other Moduli

The Montgomery method can be used for any shape of prime, although we found that the dedicated pseudo-Mersenne method was superior in all cases where it applied, but often not by much. A problem for pseudo-Mersennes is that they are intrinsically not Montgomery-friendly.

**Algorithm 3** Modular multiplication with C448 448-bit modulus, 64-bit processor

INPUT: Inputs $x, y < 2p$, radix is $2^{56}$
OUTPUT: $z = x.y \bmod 2p$

1: **function** MODMUL($x, y$)
2: $q = 2^{56}$
3: $\mathbf{t} \leftarrow x_0 y_0 \quad v_0 \leftarrow \mathbf{t} \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
4: $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=0}^{1} x_i y_{1-i} \quad v_1 \leftarrow \mathbf{t} \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
5: $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=0}^{2} x_i y_{2-i} \quad v_2 \leftarrow \mathbf{t} \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
6: $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=0}^{3} x_i y_{3-i} \quad v_3 \leftarrow \mathbf{t} \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
7: $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=0}^{4} x_i y_{4-i} \quad \mathbf{t} \leftarrow \mathbf{t} + q - v_0 \quad v_4 \leftarrow \mathbf{t} \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
8: $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=0}^{5} x_i y_{5-i} \quad \mathbf{t} \leftarrow \mathbf{t} + (q - 1 - v_1) \quad v_5 \leftarrow \mathbf{t} \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
9: $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=0}^{6} x_i y_{6-i} \quad \mathbf{t} \leftarrow \mathbf{t} + (q - 1 - v_2) \quad v_6 \leftarrow \mathbf{t} \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
10: $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=0}^{7} x_i y_{7-i} \quad \mathbf{t} \leftarrow \mathbf{t} + (q - 1 - v_3) \quad v_7 \leftarrow \mathbf{t} \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
11: $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=1}^{8} x_i y_{8-i} \quad \mathbf{t} \leftarrow \mathbf{t} + (q - 1 + v_0 - v_4) \quad v_8 \leftarrow \mathbf{t} \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
12: $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=2}^{8} x_i y_{9-i} \quad \mathbf{t} \leftarrow \mathbf{t} + (q - 1 + v_1 - v_5) \quad z_0 \leftarrow \mathbf{t} \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
13: $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=3}^{8} x_i y_{10-i} \quad \mathbf{t} \leftarrow \mathbf{t} + (q - 1 + v_2 - v_6) \quad z_1 \leftarrow \mathbf{t} \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
14: $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=4}^{8} x_i y_{11-i} \quad \mathbf{t} \leftarrow \mathbf{t} + (q - 1 + v_3 - v_7) \quad z_2 \leftarrow \mathbf{t} \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
15: $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=5}^{8} x_i y_{12-i} \quad \mathbf{t} \leftarrow \mathbf{t} + (q - 1 + v_4 - v_8) \quad z_3 \leftarrow \mathbf{t} \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
16: $\mathbf{t} \leftarrow \mathbf{t} + \sum_{i=6}^{8} x_i y_{13-i} \quad \mathbf{t} \leftarrow \mathbf{t} + (q - 1 + v_5) \quad z_4 \leftarrow \mathbf{t} \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
17: $\mathbf{t} \leftarrow \mathbf{t} + x_7 y_7 \quad \mathbf{t} \leftarrow \mathbf{t} + (q - 1 + v_6) \quad z_5 \leftarrow \mathbf{t} \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
18: $\mathbf{t} \leftarrow \mathbf{t} + (q - 1 + v_7) \quad z_6 \leftarrow \mathbf{t} \bmod q \quad \mathbf{t} \leftarrow \lfloor \mathbf{t}/q \rfloor$
19: $\mathbf{t} \leftarrow \mathbf{t} + (v_8 - 1) \quad z_7 \leftarrow t$
20: **return** $z$
21: **end function**

The NIST384 prime is particularly awkward, as it is not Montgomery-friendly on a 64-bit processor, although it is in a 32-bit setting.

Another application might be in the context of optimizing arithmetic modulo the elliptic curve group order, especially when the field is close to a power of 2. For example for C25519, the prime order group consists of $2^{252} + 27742317777372353535851937790883648493$ points, a number which exhibits multiple 0 digits.

At one time Montgomery-friendly primes of the form $2^a.3^b - 1$ were considered of interest for isogeny based cryptography [17], [12], and they may be again in the future [4].

More recently new prime moduli with very particular properties have been suggested as part of putative standards for isogeny based signature, see [18]. In a recent paper [48] alternate primes are proposed which the authors contend may lead to implementations with improved characteristics. These primes are all Montgomery friendly. For example the MFP4 prime has been suggested where $p = 3.67.2^{246} - 1$ [48].

# 10 Function inlining

One advantage of a high level language implementation is that the function calling overhead can be removed by in-lining the code. This obviously leads

to code size bloating, but can also improve performance, and hence close the performance gap against a pure assembly language implementation. As such it is a perfectly valid optimization. However, frustratingly, it can be hard to control inlining, as a compiler has its own idea of when and when not to inline a function. The compiler may, for example, be concerned to keep program code small enough to fit inside of the processor instruction cache. We could inflate our results by forcing inlining, which might appear to improve performance in the artifical context of a tight timing loop, but would actually lead to a slow down in a real-world setting. On balance we think that the fair thing to do is to merely provide a hint to the compiler, and to let it make the decision on whether or not to inline time-critical functions. The downside is that an opaque compiler decision to inline may cause some minor timing anomalies.

# 11 Comparison with pure high-level language implementations

Many existing high level language implementations of finite field arithmetic are unoptimized, and provided primarily to be correct rather than fast. Some are so-called "ref" implementations, provided only as a proof-of-concept and to demonstrate that the method works, but were never intended for production use. They may for example deliberately not exploit 128-bit integers in order to remain strictly compliant with the C standard.

But rather surprisingly many high-level implementations do exist that are deployed in the wild in real-world settings. We would speculate that this may be due to (a) an indifference to bleeding-edge performance and/or (b) a prioritisation of correctness over speed. There is some evidence for this. For example BoringSSL deploys C code generated by the Fiat Crypto project, which replaced an earlier assembly language implementation [16]. The elliptic curve cryptography used by Microsoft in the WSL2 Linux kernel [37] also uses finite field C code generated by the Fiat project, plus some derived from the verified code generated by the HACL project [57].

The Fiat and HACL projects generated high level code which provide machine checked proofs of functional correctness, an undoubted advantage. However we would suggest that the major cause of failure in finite field arithmetic would be due to limb overflow (see above), and neither of these projects prove that such failure cannot occur. As it states in [16] "we are intentionally avoiding details of the underlying machine arithmetic".

In the world of TLS, high level C implementations seem to be relatively common. A recent blog (2023) [35] describes a tailored C implementation of finite field arithmetic for the awkward NIST384 prime, which has recently been upstreamed to OpenSSL, and which claims a greater than 5 times speed up for the previous code used for digital signature. (Which suggests that for

many years users were content with an implementation 5 times slower than necessary).

| | Skylake | | Raptorlake | |
|---|---|---|---|---|
| Compiler | OpenSSL [35] | Our script | OpenSSL [35] | Our script |
| gcc | 255-219 | 181-176 | 229-196 | 165-151 |
| icx | 211-174 | 193-160 | 178-143 | 169-151 |
| clang | 225-184 | 190-159 | 199-167 | 169-147 |

Table 1: NIST384 Skylake x86-64Clock cycles mul-sqr

In table 1 we present the number of clock cycles required for both modular multiplication and squaring. It can be seen that this OpenSSL implementation seems to be quite sensitive to the choice of compiler. But also it can be seen that our script derived code does quite well in comparision.

So how does our script generated code measure up for other moduli? For the pseudo-Mersenne case the performance of the generated code is quite comparable with the earlier efforts of many others. For non-pseudo-Mersennes there is more variation.

In table 2 we compare with the Fiat-Crypto code from [16], and the C code from the PQCrypto-SIDH project [38], for a selection of moduli on our Skylake processor.

| | Fiat Crypto [16] | | PQCrypto-SIDH [38] | | Our scripts | |
|---|---|---|---|---|---|---|
| Modulus | clang | gcc | clang | gcc | clang | gcc |
| C25519 | 82-58 | 69-64 | | | 67-55 | 62-57 |
| NIST256 | 136-129 | 242-217 | | | 90-69 | 90-83 |
| NIST384 | 307-292 | 583-598 | | | 195-166 | 187-182 |
| C448 | 247-120 | 200-117 | | | 178-126 | 166-127 |
| NIST521 | 274-162 | 211-111 | | | 178-110 | 192-105 |
| SIDH434 | 558 | 927 | 1638 | 1855 | 248 | 209 |
| SIDH751 | | | 4643 | 5041 | 590 | 502 |

Table 2: Skylake x86-64 Clock cycles mul-sqr

In table 3 are the results on an ARM64 powered Raspberry PI. In this case using the ADK Karatsuba method for multiplication was the faster option for our scripts.

There is enough evidence here to allow us to express confidence that our code performance is on a par with the best high-level language implementations out there. No one compiler beats the others in all cases, and so there is a case for choosing the compiler based on the quality of the code it generates for each individual modulus. The relatively poor performance of the Fiat-Crypto implementations for the NIST256 and NIST384 moduli can be explained by their use of a saturated radix, which we suggest is a suboptimal

| | Fiat Crypto [16] | | PQCrypto-SIDH [38] | | Our scripts | |
|---|---|---|---|---|---|---|
| Modulus | clang | gcc | clang | gcc | clang | gcc |
| C25519 | 79-53 | 73-44 | | | 65-53 | 45-44 |
| NIST256 | 82-63 | 88-72 | | | 56-56 | 58-58 |
| NIST384 | 170-123 | 197-166 | | | 137-140 | 154-143 |
| C448 | 192-107 | 193-105 | | | 116-110 | 113-105 |
| NIST521 | 250-137 | 238-132 | | | 138-132 | 137-132 |
| SIDH434 | 256 | 277 | 2540 | 2784 | 213 | 229 |
| SIDH751 | | | 7316 | 7892 | 532 | 540 |

Table 3: RP5 ARM64 nanoseconds mul-sqr

strategy in this context. On the other hand for the pseudo-Mersennes our timings and those of the Fiat-crypto code are quite close.

Curiously we found that the rather obfuscated Fiat-crypto code for multiplication modulo the C25519 prime was (after some variable renaming and harmless instruction re-ordering) identical to the original venerable code from curve25519-donna-c64 [28]. Our code was also very similar, which is not really surprising given that the algorithm used is the same, and is the best one known (although see [46] for an alternate method that applies to C25519 and avoids double length register shifts).

## 12 Comparison with high-level language plus intrinsics

There is an approach to implementation which takes a middle route between full blown assembly and a high level language. The idea is to use "intrinsics" which allow direct access to assembly language instructions via inlined function calls. Often an intrinsic will result in the generation of just a single machine code instruction. These intrinsic functions are particular to each architecture, and the range and functionality available varies from one processor to another. So using intrinsics does come at the cost of portabilility. But it does permit much of the efficiency of assembly language programming without some of its overheads. For example the tricky issue of optimal register allocation is still left to the compiler.

A good example of the use of intrinsics to implement finite field arithmetic would be Pornin's crrl rust library [46]. Here saturated radix arithmetic can be used, as an intrinsic can have access to the internal carry flag. For the x86-64 architecture the rust `addcarry_u64()` intrinsic is particularly useful in the context of finite field arithmetic. However for other architectures like ARM64 and RISCV it does not exist, and must instead be implemented using less efficient non-intrinsic methods.

In tables 26 and 27 we compare our script generated code against the

intrinsic-powered crrl code for x86-64 and ARM64.

| | crrl [46] | Our scripts | |
|---|---|---|---|
| Modulus | rustc | clang | gcc |
| C25519 | 55-45 | 67-55 | 61-56 |
| C448 | 188-113 | 175-126 | 160-125 |

Table 4: Skylake x86-64Clock cycles mul-sqr

| | crrl [46] | Our scripts | |
|---|---|---|---|
| Modulus | rustc | clang | gcc |
| C25519 | 71-44 | 65-53 | 45-44 |
| C448 | 157-97 | 116-110 | 113-105 |

Table 5: RP5 ARM64 ns mul-sqr

# 13 Comparison with Assembly language

Here the aim is to answer this question: What kind of speed up can be expected from a good assembly language implementation when compared to our simple script generated C code, considering a variety of architectures and compilers? This is a particularly relevant question for the field of isogeny based cryptography, where it might be hoped that despite the large field sizes sometimes required, implementations may become viable as they might benefit because "further optimisations of the finite field arithmetic could offer substantial speed ups, as seen in the optimised assembly implementations for large characteristic implementations of SIDH" [15]. This hope may have been motivated by the somewhat misleading timings obtained from the PQCrypto-SIDH benchmarks [38], where a very slow C implementation is used for comparison.

In these tables we record the time for a modular multiplication $m_1$, the time for a modular multiplication using the ADK method $m_2$, and the time for a squaring $s$ in the form $m_1(m_2) - s$. In some cases $m_2$ and/or $s$ may be missing or not relevant. Some benchmarks provide output in clock cycles, while others provide actual timings in nanoseconds. We provide both where necessary to faciliate comparisions.

## 13.1 x86-64

Our first comparison is of the script generated C code against x86-64 assembly language implementations taken from Nath and Sarkar [43], and AWS [27].

| Source | C25519 | PM266 | PM336 | PM383 | C414 | PM512 | NIST521 |
|---|---|---|---|---|---|---|---|
| N&S-sat cyc [43] | **57-47** | 70-57 | | 90-76 | 104-90 | 138-118 | 140-119 |
| N&S-uns cyc [43] | **74-53** | 75-52 | | 119-98 | 138-118 | 193-143 | 163-126 |
| Ours-gcc cyc | 61(82)-56 | 64(86)-60 | 81(98)-61 | 118(125)-91 | 131(136)-101 | 216(207)-126 | 192(201)-105 |
| Ours-clang cyc | 67(78)-55 | 69(80)-59 | 90(105)-64 | 121(144)-86 | 135(155)-98 | 193(213)-124 | 178(198)-110 |
| Ours-icx cyc | 69(75)-52 | 65(78)-52 | 91(106)-59 | 125(151)-93 | 145(159)-100 | 208(236)-132 | 190(216)-113 |
| AWS-sat ns [27] | 28-21 | | | | | | 78-55 |
| Ours-gcc ns | 34(45)-32 | 36(48)-33 | 45(54)-34 | 66(69)-50 | 72(75)-56 | 120(115)-70 | 106(111)-58 |
| Ours-clang ns | 37(43)-31 | 38(44)-32 | 50(58)-35 | 67(80)-47 | 75(86)-54 | 107(118)-68 | 99(110)-61 |
| Ours-icx ns | 38(41)-29 | 36(43)-29 | 50(59)-33 | 69(84)-51 | 80(88)-55 | 116(131)-75 | 105(120)-63 |

Table 6: Skylake x86-64, PMs, mul-comba(mul-karatsuba)-square

| Source | NIST256 | GM266 | NIST384 | X448 | NIST521 |
|---|---|---|---|---|---|
| N&S-sat cyc [43] [42] | | | | 112-96 | 140-119 |
| N&S-uns cyc | | | | 158-116 | 163-126 |
| Ours-gcc cyc | 90(101)-83 | 84(91)-67 | 181(251)-176 | 160(165)-125 | 195(207)-170 |
| Ours-clang cyc | 90(98)-69 | 76(88)-59 | 190(206)-160 | 175(182)-126 | 203(208)-140 |
| Ours-icx cyc | 94(96)-79 | 82(93)-63 | 193(214)-159 | 173(186)-122 | 202(215)-139 |
| AWS-sat ns [27] | 29-25 | | 74-61 | | 78-55 |
| Ours-gcc ns | 50(6)-46 | 46(50)-37 | 104(139)-101 | 89(91)-69 | 108(115)-94 |
| Ours-clang ns | 50(4)-38 | 42(58)-33 | 108(114)-92 | 99(101)-70 | 113(116)-78 |
| Ours-icx ns | 52(53)-44 | 45(52)-35 | 109(119)-93 | 96(103)-68 | 112(119)-77 |

Table 7: Skylake x86-64, GMs, mul-comba(mul-karatsuba)-square

| Source | MFP4 | SIDH434 | SIDH751 | MFP4096 |
|---|---|---|---|---|
| PQCrypt cyc [38] | | 139 | 403 | |
| CSIDH cyc [13] | | | | **14758-14208** |
| Ours-gcc cyc | 70(78)-58 | 209(221)-171 | 515(502)-390 | 15684(**13729)-10712** |
| Ours-clang cyc | 75(77)-60 | 250(248)-183 | 639(590)-404 | 25113(18472)-12194 |
| Ours-icx cyc | 78(81)-55 | 246(253)-180 | 642(584)-400 | 25825(19132)-12602 |

Table 8: Skylake x86-64, MFs, mul-comba(mul-karatsuba)-square

Note that for the prime MFP4096, if using the gcc compiler our code using ADK for multiplication appears to improve on that reported in [13].

| Source | C25519 | PM266 | PM336 | PM383 | C414 | PM512 | NIST521 |
|---|---|---|---|---|---|---|---|
| N&S-sat cyc [43] | **60-46** | 75-54 | | 94-73 | 116-83 | 151-99 | 159-102 |
| N&S-uns cyc [43] | **57-36** | 57-38 | | 103-68 | 123-84 | 175-134 | 148-115 |
| Ours-gcc cyc | 70(91)-62 | 70(93)-63 | 90(99)-74 | 126(126)-91 | 139(135)-98 | 227(186)-131 | 198(177)-123 |
| Ours-clang cyc | 60(72)-58 | 68(74)-60 | 85(82)-67 | 114(122)-83 | 150(136)-88 | 195(164)-118 | 175(154)-112 |
| Ours-icx cyc | 66(70)-58 | 68(70)-55 | 86(81)-69 | 130(125)-101 | 152(128)-122 | 217(181)-145 | 195(166)-105 |
| AWS-sat ns [27] | 15-13 | 15-12 | | | | | 58-38 |
| Ours-gcc ns | 26(34)-23 | 27(35)-24 | 34(37)-28 | 48(48)-35 | 50(51)-37 | 87(71)-50 | 75(68)-47 |
| Ours-clang ns | 23(27)-22 | 26(28)-23 | 32(31)-25 | 43(46)-32 | 57(52)-34 | 74(63)-45 | 67(59)-43 |
| Ours-icx ns | 25(27)-22 | 26(26)-21 | 33(31)-26 | 50(48)-38 | 58(49)-46 | 83(69)-55 | 75(63)-40 |

Table 9: Raptorlake x86-64, PMs, mul-comba(mul-karatsuba)-square

| Source | NIST256 | GM266 | NIST384 | X448 | NIST521 |
|---|---|---|---|---|---|
| N&S-sat cyc [43] [42] | | | | 157-96 | 148-115 |
| N&S-uns cyc [43] [42] | | | | 125-98 | 157-96 |
| Ours-gcc cyc | 74(98)-68 | 70(90)-59 | 165(260)-151 | 154(163)-110 | 175(179)-124 |
| Ours-clang cyc | 85(78)-78 | 79(66)-62 | 169(191)-147 | 165(153)-126 | 192(164)-142 |
| Ours-icx cyc | 85(81)-73 | 72(67)-64 | 169(191)-151 | 156(156)-120 | 182(161)-134 |
| AWS-sat ns [27] | 18-16 | | 48-40 | | 58-38 |
| Ours-gcc ns | 28(37)-26 | 27(34)-22 | 65(99)-61 | 59(62)-42 | 67(68)-47 |
| Ours-clang ns | 33(29)-29 | 30(25)-24 | 67(73)-59 | 63(58)-48 | 73(63)-54 |
| Ours-icx ns | 32(31)-28 | 28(25)-24 | 64(73)-58 | 59(60)-46 | 69(61)-51 |

Table 10: Raptorlake x86-64, GMs, mul-comba(mul-karatsuba)-square

Observe that the Nath and Sarker binary for C25519 is identical for both Skylake and Raptorlake. On the Skylake the saturated code is faster than

| Source | MFP4 | SIDH434 | SIDH751 | MFP4096 |
|---|---|---|---|---|
| PQCrypt cyc [38] | | 147 | 390 | |
| CSIDH cyc | | | | |
| Ours-gcc cyc | 58(85)-53 | 206(212)-153 | 505(496)-373 | 15975(14349)-10135 |
| Ours-clang cyc | 82(67)-71 | 223(187)-167 | 538(437)-378 | 24206(18751)-12681 |
| Ours-icx cyc | 86(70)-62 | 210(186)-166 | 539(447)-384 | 24644(19472)-13039 |

Table 11: Raptorlake x86-64, MFs, mul-comba(mul-karatsuba)-square

the unsaturated, but on the Raptorlake this is reversed. It would appear that architectural advances have favoured the unsaturated implementation.

It would also appear that the clang and icx compilers produce better code for the ADK method than the gcc compiler.

## 13.2 ARM64

Here we compare the script generated code against ARM64 assembly language implementations, on an Apple MAC M2 and a Rasperry Pi 5.

| Source | C25519 | PM266 | PM336 | PM383 | C414 | PM512 | NIST521 |
|---|---|---|---|---|---|---|---|
| AWS-sat ns [27] | 7-5 | | | | | | 28-16 |
| Ours-gcc ns | 12(14)-11 | 12(15)-11 | 14(16)-11 | 19(20)-15 | 19(20)-16 | 30(32)-20 | 27(30)-20 |
| Ours-clang ns | 10(12)-10 | 12(12)-10 | 15(16)-10 | 19(20)-17 | 25(21)-18 | 35(27)-22 | 33(26)-18 |

Table 12: Apple M2, PMs, mul-comba(mul-karatsuba)-square

| Source | NIST256 | GM266 | NIST384 | X448 | NIST521 |
|---|---|---|---|---|---|
| AWS-sat ns [27] | 8-7 | | 21-17 | | 29-19 |
| Ours-gcc ns | 11(15)-10 | 10(13)-9 | 26(37)-24 | 23(25)-16 | 27(26)-19 |
| Ours-clang ns | 18(19)-18 | 11(11)-10 | 31(29)-26 | 24(23)-21 | 24(23)-22 |

Table 13: Apple M2, GMs, mul-comba(mul-karatsuba)-square

| Source | MFP4 | SIDH434 | SIDH751 | MFP4096 |
|---|---|---|---|---|
| PQCrypt cyc [38] | | **37** | 84 | |
| Ours-gcc ns | 10(11)-8 | 33(33)-26 | 90(72)-60 | 8684(2965)-1993 |
| Ours-clang ns | 10(9)-9 | 34(**32)-26** | 90(71)-63 | 8475(6205)-1838 |

Table 14: Apple M2, MFs, mul-comba(mul-karatsuba)-square

Observe how the AWS assembly language advantage diminishes for larger moduli. Observe also that the script generated C code on the Apple M2 is actually faster than the PQCrypto assembly language for SIDH434 and SIDH751.

| Source | C25519 | PM266 | PM336 | PM383 | C414 | PM512 | NIST521 |
|---|---|---|---|---|---|---|---|
| AWS-sat ns [27] | 36-17 | | | | | | 140-113 |
| Ours-gcc ns | 73(45)-44 | 73(49)-44 | 105(62)-61 | 172(111)-111 | 161(100)-100 | 275(170)-169 | 238(137)-132 |
| Ours-clang ns | 81(65)-53 | 75(58)-46 | 107(72)-65 | 171(114)-114 | 158(101)-99 | 277(174)-170 | 240(138)-132 |

Table 15: Raspberry pi ARM64, PMs, mul-comba(mul-karatsuba)-square

For the Raspberry Pi it is striking that the ADK Karatsuba method is particularly effective, often cutting the cost of a modular multiplication

right down to that of a modular squaring.

| Source | NIST256 | GM266 | NIST384 | X448 | NIST521 |
|---|---|---|---|---|---|
| AWS-sat ns [27] | 36-26 | | 75-61 | | 140-113 |
| Ours-gcc ns | 87(58)-58 | 73(44)-43 | 205(154)-143 | 188(113)-105 | 237(136)-131 |
| Ours-clang ns | 82(55)-56 | 73(45)-47 | 191(137)-140 | 188(116)-110 | 239(138)-142 |

Table 16: Raspberry pi ARM64, GMs, mul-comba(mul-karatsuba)-square

| Source | MFP4 | SIDH434 | SIDH751 | MFP4096 |
|---|---|---|---|---|
| PQCrypt ns [38] | | 214 | | |
| Ours-gcc ns | 87(58)-58 | 304(229)-222 | 760(540)-533 | 22476(15751)-15512 |
| Ours-clang ns | 87(59)-61 | 295(213)-218 | 760(532)-541 | 23037(16316)-15363 |

Table 17: Raspberry pi ARM64, MFs, mul-comba(mul-karatsuba)-square

### 13.3 RISCV

Unfortunately for the 64-bit RISCV architecture we did not find any assembly language implementations to compare against. But it is still of interest to observe the difference between compilers, and implementation with and without Karatsuba.

| Source | C25519 | PM266 | PM336 | PM383 | C414 | PM512 | NIST521 |
|---|---|---|---|---|---|---|---|
| Ours-gcc cyc | 125(131)-96 | 124(139)-93 | 161(172)-117 | 240(225)-185 | 288(264)-235 | 400(416)-266 | 371(396)-214 |
| Ours-clang cyc | 126(147)-97 | 132(142)-94 | 174(200)-124 | 270(297)-199 | 315(308)-204 | 465(458)-298 | 419(428)-235 |

Table 18: LicheePi RISCV, PMs, mul-comba(mul-karatsuba)-square

| Source | NIST256 | GM266 | NIST384 | X448 | NIST521 |
|---|---|---|---|---|---|
| Ours-gcc cyc | 193(180)-166 | 155(148)-129 | 431(395)-331 | 339(346)-255 | 437(391)-287 |
| Ours-clang cyc | 185(187)-155 | 163(163)-130 | 445(438)-376 | 397(382)-325 | 473(434)-345 |

Table 19: LicheePi RISCV, GMs, mul-comba(mul-karatsuba)-square

| Source | MFP4 | SIDH434 | SIDH751 | MFP4096 |
|---|---|---|---|---|
| Ours-gcc cyc | 146(140)-116 | 528(501)-411 | 1200(1136)-937 | 229179(201813)-156179 |
| Ours-clang cyc | 156(158)-131 | 559(527)-444 | 1310(1191)-915 | 84799(108006)-49635 |

Table 20: LicheePi RISCV, MFs, mul-comba(mul-karatsuba)-square

### 13.4 ARM M4

The 32-bit ARM M4 processor is very popular with cryptographers, due in large part to its very flexible range of integer multiplication instructions, which are fast and constant time. As observed in [53] Karatsuba will not be effective for the ARM M4 processor "due to the high efficiency of UMAAL instructions". As 32-bit processors are often not suitable for isogeny-based cryptography, we consider a reduced range of moduli.

In table 23 we can compare with the record setting saturated radix assembly language implementation of Aranha and Fujii [20]

| Source | C25519 | PM266 | PM336 | PM383 | C414 | PM512 | NIST521 |
|---|---|---|---|---|---|---|---|
| Ours-gcc cyc | 557-385 | 694-417 | 802-501 | 1248-796 | 1410-850 | 2462-1724 | 1790-955 |

Table 21: STM32F4 ARM M4, PMs, mul-square

| Source | NIST256 | GM266 | NIST384 | X448 | NIST521 | MFP4 |
|---|---|---|---|---|---|---|
| Ours-gcc cyc | 755-572 | 768-576 | 1625-1132 | 1639-1099 | 2155-1358 | 540-369 |

Table 22: STM32F4 ARM M4, GMS and MFs, mul-square

| Source | C25519 | PM266 | PM336 | PM383 | C414 | PM512 | NIST521 |
|---|---|---|---|---|---|---|---|
| Ours-gcc cyc | 471-343 | 569-375 | 667-420 | 1042-649 | 1157-724 | 1975-1367 | 1504-776 |
| Aranha [20] cyc | 276-252 | | | | | | |

Table 23: Teensy ARM M4, PMs, mul-square

| Source | NIST256 | GM266 | NIST384 | X448 | NIST521 | MFP4 |
|---|---|---|---|---|---|---|
| Ours-gcc cyc | 639-505 | 641-504 | 1309-963 | 1373-954 | 1722-1156 | 450-319 |

Table 24: Teensy ARM M4, GMS and MFs, mul-square

## 13.5 ARM M0

The ARM Cortex M0 processor is a popular low-powered 32-bit processor, used in the tiny Raspberry Pi Pico board. Due to its slow multiply instruction, Karatsuba works well for this processor.

| Source | C25519 | PM266 | PM336 | PM383 | C414 | PM512 | NIST521 |
|---|---|---|---|---|---|---|---|
| Ours-gcc $\mu$s | 53(36)-33 | 61(41)-37 | 81(55)-46 | 118(76)-69 | 135(86)-78 | 192(120)-111 | 179(107)-98 |

Table 25: RP2040 ARM M0, PMs, mul-comba(mul-karatsuba)-square

| Source | NIST256 | GM266 | NIST384 | X448 | NIST521 | MFP4 |
|---|---|---|---|---|---|---|
| Ours-gcc $\mu$s | 55(37)-36 | 60(38)-36 | 124(77)-71 | 148(88)-81 | 218(121)-112 | 52(35)-32 |

Table 26: RP2040 ARM M0, GMs and MFs, mul-comba(mul-karatsuba)-square

## 13.6 ESP32

The cheap ESP32 processor is very widely used in IOT applications, and is based on a RISC design. We are unaware of any cryptographic software written for it in assembly language.

| Source | C25519 | PM266 | PM336 | PM383 | C414 | PM512 | NIST521 |
|---|---|---|---|---|---|---|---|
| Ours-gcc $\mu$s | 6(6)-4 | 6(6)-4 | 9(8)-5 | 13(12)-8 | 14(13)-8 | 21(18)-12 | 19(17)-10 |

Table 27: TinyPICO ESP32, PMs, mul-comba(mul-karatsuba)-square

| Source | NIST256 | GM266 | NIST384 | X448 | NIST521 | MFP4 |
|---|---|---|---|---|---|---|
| Ours-gcc $\mu$s | 7(7)-5 | 8(7)-5 | 16(14)-11 | 17(14)-10 | 23(20)-13 | 6(6)-4 |

Table 28: TinyPICO ESP32, GMS and MFs, mul-comba(mul-karatsuba)-square

# 14 RFC7748

Here we compare the performance of our script generated high-level code compared to some state-of-the-art assembly language code, in the context of the implementation of RFC7748, specifically the shared-secret calculation, where no precomputation optimizations are available. We consider both the 64-bit x86-64, ARM64 and RISCV-64, and the 32-bit ARM M0 and M4 architectures.

The two curves supported by RFC7748 are C25519 and C448. In our case we use the pseudo-Mersenne script to generate the code for C25519 and the Montgomery script to generate the code for C448.

SIMD extensions allow parallel execution of modular arithmetic. Typically, where a regular 64-bit processor can execute a single modular multiplication, using SIMD up to four modular multiplications can be carried out in parallel, albeit commonly using four 32-bit processors. But if four independent modular multiplications can be found in a particular algorithm, then the four parallel calculations on the four 32-bit processors should complete faster than four serial calculations on a single 64-bit processor. Clearly this is dependent on the algorithm being implemented, but in the case of RFC7748 such parallellism can be found [41].

For the x86-64 Nath and Sarkar [41] have provided optimized assembly language code which makes use of the Intel AVX2 extensions to implement RFC7748. For the ARM64 the AWS team [27] has improved Lenngren's original code [30], using ideas from [1] to also provide very highly optimized RFC7748 code. The crrl rust library [46] also provides implementations of RFC7748.

To determine the improvement over our script generated code, for the x86-64 we run the code provided on our Skylake and Raptor lake processors. For the ARM64 we run the code on the RP5 and an Apple M2.

| | Skylake | | Raptorlake | |
|---|---|---|---|---|
| Modulus | C25519 | C448 | C25519 | C448 |
| Nath/Sarkar AVX2 [41] | 108881 | 398133 | 112661 | 394268 |
| Pornin crrl [46] | 146424 | 714924 | 121911 | 551947 |
| AWS [27] | (57) | | (37) | |
| Our script | 171787(95) | 809668 | 163510(63) | 657093 |

Table 29: RFC7748 x86-64 clock cycles (microseconds)

The speed-up that can be achieved by the assembly language program-

mer on the x86-64 architecture appears to rarely top times-two. And on the more modern Raptorlake architecture this advantage diminishes.

| | Apple M2 | | RP5 | |
|---|---|---|---|---|
| Modulus | C25519 | C448 | C25519 | C448 |
| Pornin crrl [46] | 29 | 123 | 137 | 560 |
| AWS [27] | 17/20 | | 127/46 | |
| Our script (ADK) | 30 | 110 | 124 | 528 |

Table 30: RFC7748 ARM64 microseconds/NEON

It is interesting to observe that for the ARM64 architecture, using NEON is clearly very benefical for the RP5, nearly 3 times faster than our C code. However for the Apple M2 the non-NEON assembly language code is faster than the NEON code. So it would appear that whereas using the NEON extensions makes a big difference for the RP5, it is suboptimal for the Apple M2. Again we can conclude that architectural developments can sometimes invalidate the assumptions on which an assembly language implementation is based.

Assembly language RFC7748 code for the RISCV-5 processor can be found here [14] and is benchmarked in table 31 . In this case there are no available SIMD extensions, and the assembly language code does not improve on the C code, which is perhaps not surprising on a reduced instruction set processor where the assembly language programmer has much less flexibility (and a compiler is less likely to run out of registers).

| | C25519 | C448 |
|---|---|---|
| Cheng et al [14] | 334741 | |
| crrl [46] | 254261 | 1626209 |
| Our script | 314621 | 1475442 |

Table 31: RFC7748 RISCV clock cycles, gcc and rustc compilers

However on 32-bit processors the situation is reversed and substantial 3-4 times speed-ups can often be achieved, albeit with considerable effort by very talented programmers. See tables 32 and 33.

| | C25519 | C448 |
|---|---|---|
| Pornin [45] | 24285 | |
| Our script (ADK) | 95054 | 437695 |

Table 32: RFC7748 RP2040 ARM M0 microseconds.

| | C25519 | C448 |
|---|---|---|
| Lenngren [31] | 617372 | |
| Our script | 1637582 | 7438431 |

Table 33: RFC7748 STM32F4 ARM M4 clock cycles

# 15 Discussion

By adopting an unsaturated radix, multiple advantages flow. Perhaps it represents an opportunity to free ourselves from the tyranny of moduli that are exact multiples of the word length. For example using an unsaturated radix a 256-bit curve will require 5 limb field elements on a 64-bit computer. If this is the case, why not use a 266-bit modulus, which is clearly more secure, but since it also requires 5 limbs we get the extra security at little extra cost. Also we allow much more flexibility in our search for suitable primes. In [3], new prime moduli are sought for elliptic curve cryptography, constrained by the assumption of a saturated radix representation and to use primes constructed as generalized Mersenne Montgomery-friendly primes from polynomials in $2^{64}$. Not many are found. However using an unsaturated radix, a pseudo-Mersenne $2^m - c$ can be chosen which has the lowest $c$, and allowing a wide range of $m$. Similarly a generalised Mersenne can be picked as $2^m - 2^k - 1$ for a range of $m$ and $k$ which are not so constrained. If one or both of $m$ and $k$ are divisible by the radix, that will certainly help.

## 15.1 Pseudo, Generalised or Friendly?

But which shape is best? An interesting head-to-head is on a 64-bit processor between the PM266 pseudo-Mersenne $2^{266} - 3$ using a radix of $2^{54}$ and the GM266 generalised Mersenne $2^{266} - 2^{168} - 1$ using a radix of $2^{56}$ bits. In both cases 5 limbs are required. For the pseudo-Mersene $2^{54}$ is chosen to avoid requiring the Nath and Sarkar "fix". For the generalised Mersenne $2^{56}$ is chosen as 56 is an exact divisor of 168. On a 64-bit processor there are a range of viable radixes that do not require an increase in the number of limbs above the bare minimum. However on a 32-bit processor we are more constrained and would recommend a radix of $2^{27}$ for the pseudo-Mersenne which again avoids the requirement for the "fix". For the generalised Mersenne we recommend a radix of $2^{28}$ as again 168 is divisible by 28. In both cases the number of limbs required is 10.

The results are mixed. As can be seen from the tables above GM266 usually beats PM266 on a 64 bit processor, but the reverse is the case for a 32-bit processor.

We observe that the MFP4 Montgomery-friendly prime is very competitive, even against C25519.

### 15.2 Saturated or unsaturated?

It would seem that an unsaturated implementation of modular multiplication for NIST256 would start at a massive disadvantage. Consider an implementation on a 64-bit processor. The saturated implementation would require the calculation and accumulation of just 16 double precision multiplications. An unsaturated implementation would require 25 such multiplication plus 15 more for the terms in the representation of the prime modulo the radix that do not simplify to 0, 1 or -1. So maybe twice as slow? In fact if we compare against a good HLL implementation using a saturated radix [44], we find that our automatically generated code is more than 50% faster.

But a saturated implementation really needs to be implemented in assembly language to demonstrate its full potential. In practise it may come down to a decision as to whether or not any speed-up that arises from using an assembly language saturated radix implementation is worth the substantial extra effort in the context of portability, maintainability, and confidence in the correctness of the implementation.

### 15.3 Assembly or High level language?

The answer to this question appears to be heavily dependent on the computer architecture. In all cases it would appear that for modular multiplication the choice comes down to (a) an assembly language implementation using a saturated radix, (b) an assembly language implementation using an unsaturated radix, (c) a HLL implementation using an unsaturated radix.

Consider the saturated and unsaturated implementations provided by Nath and Sarkar [43] for the x86-64 architecture. Here we can make an observation. In Tables 4 and 5 in [43] there is an entry for NIST521, which shows a modest improvement over the previous record. However that previous record was written entirely in C [22]. Which begs the question: Just how much of a speed up can be achieved by making the non-trivial decision to implement in assembly? It is this question that this paper has attempted to answer.

We have made comparisons with our automatically generated code which uses essentially the same algorithm for pseudo-Mersennes, as described above. Any comparison between saturated and unsaturated is heavily dependent on whether or not an extra limb is required for the unsaturated implementation. For example the prime $2^{512} - 569$ requires an extra 9th limb for an unsaturated implementation, whereas the prime $2^{521} - 1$ does not.

We let the reader draw their own conclusions. Suffice it to say that the appreciable extra effort required for an assembly language implementation often appears to provide at best a 50% speed up. And actually getting that reward can be difficult.

However it would be a mistake to try and extrapolate the x86-64 experience to other architectures. The 32-bit ARM M4 instruction set has particularities that reward the determined assembly language programmer, with twice as many registers and a flexible range of fast integer multiplication instructions. Furthermore compiler register allocation is often very poor [54]. When compared against Lenngren's implementation of NIST256 for the 32-bit ARM M4 [32] we find that our Montgomery code is almost 3 times slower. See also [20] for a fast saturated radix implementation of C25519 modular multiplication in M4 assembly language.

The RISC-V architecture is quite different again, as noted by Pornin [47]. Due to the lack of a carry flag in its architecture, an unsaturated implementation will most likely always be preferred [55].

One tentative general conclusion might be that, if using an unsaturated radix, then there is probably little to be gained from re-writing the code in assembly language. Another conclusion might be that assembly language implementation works best for smaller moduli, particularly on 32-bit processors.

An issue with an assembly language implementation is the phenomenon known as "software rot". Over time architectures change, and compilers can take advantage of that. An assembly language implementation cannot adapt so easily – it is very much of its time.

## References


[1] A. Abdulrahman, H. Becker, M. Kannwischer, and F. Klein. Fast and clean: Auditable high-performance assembly via constraint solving. Cryptology ePrint Archive, Paper 2022/1303, 2022. `https://eprint.iacr.org/2022/1303`.

[2] D. F. Aranha, C. P. L. Gouvea, T. Markmann, R. S. Wahby, and K. Liao. RELIC is an Efficient LIbrary for Cryptography. `https://github.com/relic-toolkit/relic`.

[3] Jean-Claude Bajard and Sylvain Duquesne. Montgomery-friendly primes and applications to cryptography. *Journal of Cryptographic Engineering*, 11:399 – 415, 2021.

[4] A. Basso. Poke: A framework for efficient pkes, split kems, and oprfs from higher-dimensional isogenies. Cryptology ePrint Archive, Paper 2024/624, 2024. `https://eprint.iacr.org/2024/624`.

[5] D. J. Bernstein. Irrelevant patents on elliptic-curve cryptography. `https://cr.yp.to/ecdh/patents.html`.

[6] D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In *PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–208. Springer, 2006.

[7] D. J. Bernstein. libcpucycles, 2024. `https://cpucycles.cr.yp.to/`.

[8] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B. Yang. High-speed high-security signatures. *J. Cryptographic Engineering*, 2(2):77–89, 2012.

[9] J. Bos, T. Kleinjung, and D. Page. Efficient modular multiplication. Cryptology ePrint Archive, Paper 2021/1151, 2021. `https://eprint.iacr.org/2021/1151`.

[10] J. Bos and P. Montgomery. *Montgomery Arithmetic from a Software Perspective*. Cambridge University Press, 2017.

[11] J. Bos, M. Naehrig, P. Longa, and C. Costello. Selecting elliptic curves for cryptography: an efficiency and security analysis. *Journal of Cryptographic Engineering*, 2015.

[12] Joppe W. Bos and Simon Friedberger. Fast arithmetic modulo $2^x p^y \pm 1$. Cryptology ePrint Archive, Paper 2016/986, 2016. `https://eprint.iacr.org/2016/986`.

[13] F. Campos, J. Chavez-Saab, J. Chi-Dominguez, M. Meyer, K. Reijnders, F. Rodriguez-Henriquez, P. Schwabe, and T. Wiggers. Optimizations and practicality of high-security CSIDH. *IACR Communications in Cryptology*, 1, 2024.

[14] H .Cheng. RVKEX: ECC/isogeny-based key exchange on 64-bit RISC-V, 2024. `https://github.com/scarv/rvkex`.

[15] P. Dartois, L. Maino, G. Pope, and D. Robert. An algorithmic approach to $(2, 2)$-isogenies in the theta model and applications to isogeny-based cryptography. Cryptology ePrint Archive, Paper 2023/1747, 2023. `https://eprint.iacr.org/2023/1747`.

[16] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1202–1219, 2019.

[17] Armando Faz-Hernandez, Julio Lopez, Eduardo Ochoa-Jimenez, and Francisco Rodriguez-Henriquez. A faster software implementation of the supersingular isogeny diffie-hellman key exchange protocol. *IEEE Transactions on Computers*, 67(11):1622–1636, 2018.

[18] L. De Feo, D. Kohel, A. Leroux, C. Petit, and B. Wesolowski. SQISign: Compact post-quantum signatures from quaternions and isogenies. In *Asiacrypt 2020*, volume 12491 of *Lecture Notes in Computer Science*, pages 64–93. Springer, 2020.

[19] J. Fried, P. Gaudry, N. Heninger, and E. Thome. A kilobit hidden SNFS discrete logarithm computation. Cryptology ePrint Archive, Paper 2015/625, 2016. `https://eprint.iacr.org/2016/961`.

[20] H. Fujii and D. F. Aranha. Curve25519 for the Cortex-M4 and beyond. In *Progress in Cryptology – LATINCRYPT 2017*, pages 109–127. Springer International Publishing, 2019.

[21] D. M. Gordon. Discrete logarithms in $\mathbb{F}_p$ using the number field sieve. *SIAM J. Discrete Math*, 6(1):124–138, 1993.

[22] Robert Granger and Michael Scott. Faster ECC over $\mathbb{F}_{2^{521}-1}$. In *PKC 2015*, volume 9020 of *Lecture Notes in Computer Science*, pages 539–553. Springer, 2015.

[23] Shay Gueron and Vlad Krasnov. Fast prime field elliptic-curve cryptography with 256-bit primes. *Journal of Cryptographic Engineering*, 5:141–151, 06 2014.

[24] Tim Guneysu and Christof Paar. Ultra high performance ECC over NIST primes on commercial FPGAs. In *Cryptographic Hardware and Embedded Systems - CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings*, volume 5154 of *Lecture Notes in Computer Science*, pages 62–78. Springer, 2008.

[25] Mike Hamburg. Ed448-Goldilocks, a new elliptic curve. Cryptology ePrint Archive, Paper 2015/625, 2015. `https://eprint.iacr.org/2015/625`.

[26] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic CUrve Cryptography*. Springer, 2003.

[27] Amazon Web Services Labs. s2n-bignum, 2024. `https://github.com/awslabs/s2n-bignum`.

[28] A. Langley. curve25519-donna, 2008. `https://github.com/agl/curve25519-donna/`.

[29] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic Curves for Security. RFC 7748, 2016.

[30] E. Lenngren. Aarch64 optimized implementation for x25519, 2019. `https://github.com/Emill/X25519-AArch64/blob/master/X25519_AArch64.pdf`.

[31] Emil Lenngren. X25519 for arm cortex-m4 and other arm processors, 2020. https://github.com/Emill/X25519-Cortex-M4.

[32] Emil Lenngren. P256-cortex-m4, 2021. https://github.com/Emill/P256-Cortex-M4/blob/master/p256-cortex-m4-asm-gcc.S.

[33] M. Lochter and J. Merkle. Elliptic curve cryptography (ECC) Brainpool standard curves and curve generation. RFC 5639, 2010.

[34] Michael B. McLoughlin. addchain: Cryptographic addition chain generation in go. Repository https://github.com/mmcloughlin/addchain, 2021.

[35] R. McLure. Going out on a limb: Efficient elliptic curve arithmetic in OpenSSL. https://sthbrx.github.io/blog/2023/08/07/going-out-on-a-limb-efficient-elliptic-curve-arithmetic-in-openssl/.

[36] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*. CRC Press, 1996.

[37] Microsoft. Wsl2 linux kernel, 2019. https://github.com/microsoft/WSL2-Linux-Kernel/blob/linux-msft-wsl-5.15.y/lib/crypto/curve25519-hacl64.c.

[38] Microsoft. Sidh v3.5.1, 2022. https://github.com/microsoft/PQCrypto-SIDH.

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

[40] N. El Mrabet and M. Joye, editors. *Guide to Pairing-Based Cryptography*. Chapman and Hall/CRC, 2016.

[41] K. Nath and P. Sarkar. Efficient 4-way vectorizations of the montgomery ladder. Cryptology ePrint Archive, Paper 2020/378, 2020. https://eprint.iacr.org/2020/378.

[42] K. Nath and P. Sarkar. Reduction modulo $2^{448} - 2^{224} - 1$. *Mathematical Cryptology*, 0(1):8–21, 2021.

[43] Kaushik Nath and Palash Sarkar. Efficient arithmetic in pseudo-Mersenne prime order fields. *Advances in Mathematics of Communications*, 16(2):303–348, 2022.

[44] T. Pornin. BearSSL - a smaller SSL/TLS library. https://bearssl.org/gitweb/?p=BearSSL;a=blob;f=src/ec/ec_p256_m64.c.

[45] T. Pornin. X25519 implementation for ARM cortex-M0/M0+, 2020. https://github.com/pornin/x25519-cm0.

[46] T. Pornin. crrl: Rust library for cryptographic research, 2023. https://github.com/pornin/crrl.

[47] T. Pornin. On multiplications with unsaturated limbs, 2023. https://research.nccgroup.com/2023/09/18/on-multiplications-with-unsaturated-limbs/.

[48] M. Corte-Real Santos, J. Komada Eriksen, M. Meyer, and K. Reijnders. ApresSQI: Extra fast verification for SQIsign using extension-field signing. Cryptology ePrint Archive, Paper 2023/1559, 2023.

[49] Mike Scott. Ed3363 (highfive) – an alternative elliptic curve. Cryptology ePrint Archive, Paper 2015/991, 2015. https://eprint.iacr.org/2015/991.

[50] Mike Scott. Missing a trick: Karatsuba variations. Cryptology ePrint Archive, Paper 2015/1247, 2015.

[51] Mike Scott. Slothful reduction. Cryptology ePrint Archive, Paper 2017/437, 2017.

[52] Mike Scott. A note on the calculation of some functions in finite fields: Tricks of the trade. Cryptology ePrint Archive, Paper 2020/1497, 2020.

[53] H. Seo and R. Azarderakhsh. Curve448 on 32-bit ARM costex-M4. In *ICISC 2020*, pages 125–139, 2020.

[54] Ko Stoffelen. Instruction scheduling and register allocation on ARM Cortex-M, 2016. https://repository.ubn.ru.nl/bitstream/handle/2066/166113/166113.pdf.

[55] Ko Stoffelen. Efficient cryptography on the RISC-V architecture. Cryptology ePrint Archive, Paper 2019/794, 2019. https://eprint.iacr.org/2019/794.

[56] David Wong. A readable specification of TLS 1.3, 2018. https://www.davidwong.fr/tls13/.

[57] J. Zinzindohouee, K. Bhargavan, J. Protzenko, and B. Beurdouche. Hacl*: A verified modern cryptographic library. Cryptology ePrint Archive, Paper 2015/625, 2015. https://eprint.iacr.org/2017/536.