Generalised Mersenne Numbers Revisited

Robert Granger[1⋆] and Andrew Moss[2]

[1] Claude Shannon Institute, University College Dublin, Ireland
rgranger@computing.dcu.ie
[2] Blekinge Institute of Technology, Sweden
awm@bth.se

Abstract. Generalised Mersenne Numbers (GMNs) were defined by Solinas in 1999 and feature in the NIST Digital Signature Standard (FIPS 186-2) for use in elliptic curve cryptography. Their form is such that modular reduction is extremely efficient, thus making them an attractive choice for modular multiplication implementation. However, the issue of residue multiplication efficiency seems to have been overlooked. Asymptotically, using a cyclic rather than a linear convolution, residue multiplication modulo a Mersenne number is twice as fast as integer multiplication; this property does not hold for prime GMNs, unless they are of Mersenne's form. In this work we exploit an alternative generalisation of Mersenne numbers for which an analogue of the above property — and hence the same efficiency ratio — holds, even at bitlengths for which schoolbook multiplication is optimal, while also maintaining very efficient reduction. Moreover, our proposed primes are abundant at any bitlength, whereas GMNs are extremely rare. Our multiplication and reduction algorithms can also be easily parallelised, making our arithmetic particularly suitable for hardware implementation. Furthermore, the field representation we propose also naturally protects against side-channel attacks, including timing attacks, simple power analysis and differential power analysis, which is essential in many cryptographic scenarios, in constrast to GMNs.

Keywords: Prime fields, high-speed arithmetic, elliptic curve cryptography, generalised Mersenne numbers, cyclotomic primes, generalised repunit primes

1 Introduction

The problem of how to efficiently perform arithmetic in $\mathbb{Z}/N\mathbb{Z}$ is a very natural one, with numerous applications in computational mathematics and number theory, such as primality proving [1], factoring [39], and coding theory [61], for example. It is also of central importance to nearly all public-key cryptographic systems, including the Digital Signature Algorithm [21], RSA [47], and elliptic curve cryptography (ECC) [33,41]. As such, from both a theoretical and a practical perspective it is interesting and essential to have efficient algorithms for working in this ring, for either arbitrary or special moduli, with the application determining whether generality (essential for RSA for instance), or efficiency (desirable for ECC) takes precedence.

Two intimately related factors need consideration when approaching this problem. First, how should one represent residues? And second, how should one perform arithmetic on these representatives? A basic answer to the first question is to use the canonical representation $\mathbb{Z}/N\mathbb{Z} = \{0, \ldots, N-1\}$. With regard to modular multiplication for example, an obvious answer to the second question is to perform integer multiplication of residues, followed by reduction of the result modulo N, in order to obtain a canonical representative once again. Using this approach, the two components needed for efficient modular arithmetic are clearly fast integer arithmetic, and fast modular reduction.

⋆ Supported by the Claude Shannon Institute, Science Foundation Ireland Grant No. 06/MI/006.

At bitlengths for which schoolbook multiplication is optimal, research on fast modular multiplication has naturally tended to focus on reducing the cost of the reduction step. For arbitrary moduli, Montgomery's celebrated algorithm [42] enables reduction to be performed for approximately the cost of a residue by residue multiplication. For the Mersenne numbers $M_k = 2^k - 1$, efficient modular multiplication consists of integer residue multiplication to produce a $2k$-bit product $U \cdot 2^k + L$, with U, L of at most k-bits, followed by a single modular addition $U + L \bmod M_k$ to effect the reduction, as is well known. In 1999 Solinas proposed an extension of this method to a larger class of integers: the Generalised Mersenne Numbers (GMNs) [53]. As they are a superset, GMNs are more numerous than the Mersenne numbers and hence contain more primes, yet incur little additional overhead in terms of performance [11]. In 2000, NIST recommended ten fields for use in the ECDSA: five binary fields and five prime fields, and due to their performance characteristics the latter of these are all GMNs [21], which range from 192 to 521 bits in size.

For the GMNs recommended by NIST, there is no interplay between the residue multiplication and reduction algorithms, each step being treated separately with respect to optimisation. On the other hand, at asymptotic bitlengths the form of the modulus may be effectively exploited to speed up the residue multiplication step. For the Mersenne numbers M_k in particular, modular multiplication can be performed for any k using a cyclic convolution effected by a discrete weighted transform [16, §3.1]. As such, multiplication modulo Mersenne numbers is approximately twice as fast as multiplication of integers of the same bitlength, for which a linear convolution is required, as each multiplicand must be padded with k zeros before a cyclic convolution of length $2k$ can be performed. For Montgomery multiplication at asymptotic bitlengths, the reduction step can be made 25% cheaper, again by using a cyclic rather than a linear convolution for one of the required multiplications [46]. However, since the multiplication step is oblivious to the form of the modulus, it seems unlikely to possess the same efficiency benefits that the Mersenne numbers enjoy. These considerations raise the natural question of whether there exists a similar residue multiplication speed up at bitlengths for which schoolbook multiplication is optimal? Certainly for the modulus $N = 2^k$, such a speed up can be achieved, since the upper half words of the product can simply be ignored. However, this modulus is unfortunately not at all useful for ECC.

In this work we answer the above question affirmatively, using an alternative generalisation of Mersenne numbers, which has several desirable features:

- **Simple.** Our proposed family is arguably a far more natural generalisation of Mersenne numbers than Solinas', and gives rise to beautiful multiplication and reduction algorithms.
- **Abundant.** Our primes are significantly more numerous than the set of prime GMNs and are abundant for all tested bitlengths; indeed their number can be estimated using Bateman and Horn's quantitative version [3] of Schinzel and Sierpiński's "Hypothesis H" [49].
- **Fast multiplication.** Our residue multiplication is nearly twice as fast as multiplication of integer residues.
- **Fast reduction.** Our reduction has linear complexity and is particularly efficient for specialised parameters, although such specialisation comes at the cost of reducing the number of primes available.
- **Parallelisable.** Both multiplication and reduction can be easily parallelised, making our arithmetic particularly suitable for hardware implementation.
- **Side-channel secure.** Our representation naturally protects against well-known side-channel attacks on ECC (see [10, ch. IV] for an overview), in contrast to the NIST GMNs,

see [48] and [51, §3.2]. This includes timing attacks [35, 57], simple power analysis [48] and differential power analysis [36].

This article provides an introductory (and comprehensive) theoretical framework for the use of our proposed moduli. It thus serves as a foundation for a new approach to the secure and efficient implementation of prime fields for ECC, both in software and in hardware. At a high level, our proposal relies on the combination of a remarkable algebraic identity used by Nogami, Saito, and Morikawa in the context of extension fields [44], together with the residue representation and optimisation of the reduction method proposed by Chung and Hasan [14], which models suitable prime fields as the quotient of an integer lattice by a particular equivalence relation. To verify the validity of our approach, we also provide a proof-of-concept implementation that is already competitive with the current fastest modular multiplication algorithms at contemporary ECC security levels [5, 6, 22, 23, 27, 40].

The sequel is organised as follows. In §2 we present some definitions and recall related work. In §3 we describe the basis of our arithmetic, then in §4-6 we present details of our residue multiplication, reduction and representation respectively. In §7 we show how to ensure I/O stability for modular multiplication, then in §8 we put everything together into a full modular multiplication algorithm. We then address other arithmetic operations and give a brief treatment of side-channel secure ECC in §9, and in §10 show how to generate suitable parameters. In §11 we present our implementation results and finally, in §12 we draw some conclusions.

2 Definitions and Related Work

In this section we introduce the cyclotomic primes and provide a summary of related work. We begin with the following definition.

Definition 1. *For $n \geq 1$ let ζ_n be a primitive n-th root of unity. The n-th cyclotomic polynomial is defined by*

$$\Phi_n(x) = \prod_{(k,n)=1} (x - \zeta_n^k) = \prod_{d|n} (1 - x^{n/d})^{\mu(d)},$$

where μ is the Möbius function.

Two basic properties of the cyclotomic polynomials are that they have integer coefficients, and are irreducible over $\mathbb{Z}$. These two properties ensure that the evaluation of a cyclotomic polynomial at an integer argument will also be an integer, and that this integer will not inherit a factorisation from one in $\mathbb{Z}[x]$. One can therefore ask whether or not these polynomials ever assume prime values at integer arguments, which leads to our next definition.

Definition 2. *For $n \geq 1$ and $t \in \mathbb{Z}$, if $p = \Phi_n(t)$ is prime, we call p an n-th cyclotomic prime, or simply a cyclotomic prime.*

Note that for all primes p, we have $p = \Phi_1(p + 1) = \Phi_2(p - 1)$, and so trivially all primes are cyclotomic primes. These instances are also trivial in the context of the algorithms we present for performing arithmetic modulo these primes, since in both cases the cyclotomic polynomials are linear and our algorithms reduce to ordinary Montgomery arithmetic. Hence for the remainder of the article we assume $n \geq 3$.

In addition to being prime-evaluations of cyclotomic polynomials, note that for a cyclotomic prime $p = \Phi_n(t)$, the field $\mathbb{F}_p$ can be modelled as the quotient of the ring of integers of the n-th cyclotomic field $\mathbb{Q}(\zeta_n)$, by the prime ideal $\pi = \langle p, \zeta_n - t \rangle$. This is precisely how one would represent $\mathbb{F}_p$ when applying the Special Number Field Sieve to solve discrete logarithms in $\mathbb{F}_p$, for example [38]. Hence our nomenclature for these primes seems apt. This interpretation of $\mathbb{F}_p$ for p a cyclotomic prime is implicit within the arithmetic we develop here, albeit only insofar as it provides a theoretical context for it; this perspective offers no obvious insight into how to perform arithmetic efficiently and the algorithms we develop make no use of it at all. Similarly, the method of Chung and Hasan [14] upon which our residue representation is based can be seen as arising in exactly the same way for the much larger set of primes they consider, with the field modelled as a quotient of the ring of integers of a suitable number field by a degree one prime ideal, just as for the cyclotomic primes.

2.1 Low redundancy cyclotomic primes

The goal of the present work is to provide efficient algorithms for performing $\mathbb{F}_p$ arithmetic, for $p = \Phi_n(t)$ a cyclotomic prime. As will become clear from our exposition, in order to exploit the available cyclic structure — for both multiplication and reduction — we do not use the field $\mathbb{Z}/\Phi_n(t)\mathbb{Z}$, but instead embed into the slightly larger ring $\mathbb{Z}/(t^n - 1)\mathbb{Z}$ if n is odd, and $\mathbb{Z}/(t^{n/2} + 1)\mathbb{Z}$ if n is even. In each case, using the larger ring potentially introduces an expansion factor $e(n)$ into the residue representation. One can alternatively view this in terms of a redundancy measure $r(n)$, where $r = e - 1$. Since using a larger ring for arithmetic will potentially be slower, we now identify three families of cyclotomic polynomials for which the above embeddings have low redundancy.

For n even, there is a family of cases for which the above embedding does not introduce any redundancy, namely for $n = 2^k$, since $\Phi_{2^k}(t) = t^{2^{k-1}} + 1 = t^{2^k/2} + 1$, and hence $e = 1$ and $r = 0$. When $t = 2$ these are of course the Fermat numbers, and for general t these integers are known as Generalised Fermat Numbers (GFNs). It is expected that for each k there are infinitely many t for which $t^{2^k} + 1$ is prime [18, §3].

If $n = 2p$ for p prime, then $\Phi_{2p}(t) = t^{p-1} - t^{p-2} + \cdots + t - 1 = (t^p + 1)/(t + 1)$ and in this case $e = p/(p - 1)$ and $r = 1/(p - 1)$. The primality of these numbers was studied in [19], and while they apparently do not have a designation in the literature, one can see that by substituting t with $-t$ in the third family below produces this one. For general even n we have $e = n/2\phi(n)$ and $r = (n - 2\phi(n))/2\phi(n)$, with $\phi(\cdot)$ Euler's totient function, which is the degree of $\Phi_n(x)$. Hence amongst those even n which are not a power of 2, this family produces the successive local minima of r.

For odd n, we have $e = n/\phi(n)$ and $r = (n - \phi(n))/\phi(n)$. The successive local minima of r occur at $n = p$ for p prime, in which case $\Phi_p(t) = t^{p-1} + t^{p-2} + \cdots + t + 1 = (t^p - 1)/(t - 1)$, also with $r = 1/(p-1)$. When $t = 2$ these are of course the Mersenne numbers, and in analogy with the case of Fermat numbers, it would be natural to refer to these integers for general t as Generalised Mersenne Numbers, particularly as one can show they share the aforementioned asymptotic efficiency properties of the Mersenne numbers, while Solinas' GMNs do not, unless they are of Mersenne's form. However, this family of numbers is known in the literature as generalised repunits [17,52,58], since their base-t expansion consists entirely of 1's. Therefore for the sake of uniform nomenclature, we use the following definition.

Definition 3. *For $m + 1$ an odd prime let*

$$p = \Phi_{m+1}(t) = t^m + t^{m-1} + \cdots + t + 1.$$

We call such an integer a Generalised Repunit*; when p is prime we call it a* Generalised Repunit Prime (GRP)*.*

We have developed modular multiplication algorithms for both GRPs and GFNs. In terms of efficiency, for GRPs and GFNs of the same bitlength the respective multiplication algorithms require exactly the same number of word-by-word multiplications. Also, our reduction algorithms for both GRPs and GFNs are virtually identical. However, the multiplication algorithm for GFNs is far less elegant, is not perfectly parallelisable and contains more additions. Furthermore, for a given bitlength there are fewer efficient GFN primes than there are GRPs — as the bitlength of GFNs doubles as k is incremented — and the I/O stability analysis for multiplication modulo a GRP is far simpler. Therefore in this exposition we focus on algorithms for performing arithmetic modulo GRPs and their analysis only. Note that the studies of GRPs [17,58] consider only very small t and large m, whereas we will be interested in t approximately the word base of the target architecture, and m the number of words in the prime whose field arithmetic we are to implement. Hence one expects (and finds) there to be very many GRPs for any given relevant bitlength, see §10.

2.2 Related work

In the context of extension fields, let $m+1$ be prime and let p be a primitive root modulo $m+1$. Then $\mathbb{F}_{p^m} = \mathbb{F}_p[x]/(\Phi_{m+1}(x)\mathbb{F}_p[x])$. In the binary case, i.e., $p = 2$, several authors have proposed the use of this polynomial — also known as the all-one polynomial (AOP) — to obtain efficient multiplication algorithms [9,29,50,59]. All of these rely on the observation that the field $\mathbb{F}_2[x]/(\Phi_{m+1}(x)\mathbb{F}_2[x])$ embeds into the ring $\mathbb{F}_2[x]/((x^{m+1}+1)\mathbb{F}_2[x])$ — referred to by Silverman [50] as the "ghost bit" basis — which possesses a particularly nice cyclic structure, but introduces some redundancy. Similarly, this idea applies to any cyclotomic polynomial, and several authors have investigated this strategy, embedding suitably defined extension fields into the ring $\mathbb{F}_2[x]/((x^n+1)\mathbb{F}_2[x])$ [20,24,60].

For odd characteristic extension fields, Silverman noted that the "ghost bit" basis for $p = 2$ extends easily to larger p [50], while Kwon *et al.* have explored this idea further [37]. Central to our application is the work of Nogami, Saito and Morikawa [44], who used the AOP to obtain a very fast multiplication algorithm, see §4. The use of cyclotomic polynomials in extension field arithmetic is therefore well studied. In the context of prime fields however, the present work appears to be the first to transfer ideas for cyclotomic polynomials from the domain of extension field arithmetic to prime field arithmetic, at least for the relatively small bitlengths for which schoolbook multiplication is optimal.

With regard to the embedding of a prime field into a larger integer ring, the idea of operand scaling was introduced by Walter in order to obtain a desired representation in the higher-order bits [54], which aids in the estimation of the quotient when using Barrett reduction [2]. Similarly, Ozturk *et al.* proposed using fields with characteristics dividing integers of the form $2^k \pm 1$, with particular application to ECC [45]. As stated in the introduction, there are numerous very efficient prime field ECC implementations [5,6,23,27,40]. While the moduli used in these instances permit fast reduction algorithms, and the implementations are highly optimised, it would appear that none of them permit the same residue multiplication speed up that we present here, which is one of the central distinguishing features of the present work.

3 GRP Field Representation

In this section we present a sequence of representations of $\mathbb{F}_p$, with p a GRP, the final one being the target representation which we use for our arithmetic. We recall the mathematical framework of Chung-Hasan arithmetic, in both the general setting and as specialised to GRPs, focusing here on the underlying theory, deferring explicit algorithms for residue multiplication, reduction and representation until §4-6.

3.1 Chung-Hasan arithmetic

We now describe the ideas behind Chung-Hasan arithmetic [12–14]. The arithmetic was developed for a class of integers they term low-weight polynomial form integers (LWPFIs), whose definition we now recall.

Definition 4. *An integer p is a low-weight polynomial form integer (LWPFI), if it can be represented by a monic polynomial $f(t) = t^n + f_{n-1}t^{n-1} + \cdots + f_1 t + f_0$, where t is a positive integer and $|f_i| \leq \xi$ for some small positive integer $\xi < t$.*

Note that if for a given LWPFI each $f_i \in \{\pm 1, 0\}$ and $t = 2^k$, then it is a GMN, as defined by Solinas [53]. The key idea of Chung and Hasan is to perform arithmetic modulo p using representatives from the polynomial ring $\mathbb{Z}[T]/(f(T)\mathbb{Z}[T])$. To do so, one uses the natural embedding $\psi : \mathbb{F}_p \hookrightarrow \mathbb{Z}[T]/(f(T)\mathbb{Z}[T])$ obtained by taking the base t expansion of an element of $\mathbb{F}_p$ in the canonical representation $\mathbb{F}_p = \{0, \ldots, p-1\}$, and substituting T for t. To compute ψ^{-1} one simply makes the inverse substitution and evaluates the expression modulo p.

The reason for using this ring is straightforward: since ψ^{-1} is a homomorphism, when one computes $z(T) = x(T) \cdot y(T)$ in $\mathbb{Z}[T]$, reducing the result modulo $f(T)$ to give $w(T)$ does not change the element of $\mathbb{F}_p$ represented by $z(T)$, i.e., if $z(T) \equiv w(T) \pmod{f(T)}$, then $z(t) \equiv w(t) \pmod{p}$, since $p = f(t)$. Furthermore, since $f(T)$ has very small coefficients, $w(T)$ can be computed from $z(T)$ using only additions and subtractions. Hence given the degree $2(n-1)$ product of two degree $n-1$ polynomials in $\mathbb{Z}[T]$, its degree $n-1$ representation in $\mathbb{Z}[T]/(f(T)\mathbb{Z}[T])$ can be computed very efficiently. Note that for non-low-weight polynomials this would no longer be the case.

The only problem with this approach is that when computing $z(T)$ as above, the coefficients of $z(T)$, and hence $w(T)$, will be approximately twice the size of the inputs' coefficients, and if further operations are performed the representatives will continue to expand. Since for I/O stability one requires that the coefficients be approximately the size of t after each modular multiplication or squaring, one must somehow reduce the coefficients of $w(T)$ to obtain a standard, or reduced representative, while ensuring that $\psi^{-1}(w(T))$ remains unchanged.

Chung and Hasan refer to this issue as the *coefficient reduction problem* (CRP), and developed three solutions in their series of papers on LWPFI arithmetic [12–14]. Each of these solutions is based on an underlying lattice, although this was only made explicit in [14]. Since the lattice interpretation is the most elegant and simplifies the exposition, in the sequel we opt to develop the necessary theory for GRP arithmetic in this setting.

3.2 Chung-Hasan representation for GRPs

Let $p = \Phi_{m+1}(t)$ be a GRP. Our goal is to develop arithmetic for $\mathbb{F}_p$, and we begin with the canonical representation $\mathbb{F}_p = \mathbb{Z}/\Phi_{m+1}(t)\mathbb{Z}$. As stated in §2.1, the first map in our chain

of representations takes the canonical ring and embeds it into $\mathbb{Z}/(t^{m+1}-1)\mathbb{Z}$, for which the identity map suffices. To map back, one reduces a representative modulo p. We then apply the Chung-Hasan transformation of §3.1, which embeds the second ring into $\mathbb{Z}[T]/(T^{m+1}-1)\mathbb{Z}[T]$, by taking the base t expansion of a canonical residue representative in $\mathbb{Z}/(t^{m+1}-1)\mathbb{Z}$, and substituting T for t. We call this map ψ. To compute ψ^{-1} one simply makes the inverse substitution and evaluates the expression modulo $t^{m+1}-1$.

Note that the codomain of ψ may be regarded as an $(m+1)$-dimensional vector space over $\mathbb{Z}$, equipped with the natural basis $\{T^m, \ldots, T, 1\}$. In particular, for $x(T) \in \mathbb{Z}[T]/(T^{m+1}-1)\mathbb{Z}[T]$, where

$$x(T) = x_m T^m + \ldots + x_1 T + x_0,$$

one can consider $x(T)$ to be a vector $\overline{\mathbf{x}} = [x_m, \ldots, x_0] \in \mathbb{Z}^{m+1}$. Since $\mathbb{Z}^{m+1}$ has elements whose components are naturally unbounded, for each $x \in \mathbb{Z}/(t^{m+1}-1)\mathbb{Z}$ there are infinitely many elements of $\mathbb{Z}^{m+1}$ that map via ψ^{-1} to x. Therefore in order to obtain a useful isomorphism directly between $\mathbb{Z}/(t^{m+1}-1)\mathbb{Z}$ and $\mathbb{Z}^{m+1}$, we identify two elements of $\mathbb{Z}^{m+1}$ whenever they map via ψ^{-1} to the same element of $\mathbb{Z}/(t^{m+1}-1)\mathbb{Z}$, i.e.,

$$\overline{\mathbf{x}} \sim \overline{\mathbf{y}} \iff \psi^{-1}(\overline{\mathbf{x}}) \equiv \psi^{-1}(\overline{\mathbf{y}}) \pmod{t^{m+1}-1}, \tag{3.1}$$

and take the image of ψ to be the quotient of $\mathbb{Z}^{m+1}$ by this equivalence relation. Pictorially, we thus have:

$$\mathbb{F}_p \subset \mathbb{Z}/(t^{m+1}-1)\mathbb{Z} \cong \mathbb{Z}^{m+1}/\sim$$

As mentioned in §3.1, for each coset in $\mathbb{Z}^{m+1}/\sim$, we should like to use a minimal, or in some sense 'small' representative, in order to facilitate efficient arithmetic after a multiplication or a squaring, for example. Since we know that the base-t expansion of every $x \in \mathbb{Z}/(t^{m+1}-1)\mathbb{Z}$ gives one such representative for each coset in $\mathbb{Z}^{m+1}/\sim$, for a reduction algorithm we just need to be able to find it, or at least one whose components are of approximately the same size. Chung and Hasan related finding such 'nice' or reduced coset representatives to solving a computational problem in an underlying lattice, which we now recall.

3.3 Lattice interpretation

Given an input vector $\overline{\mathbf{z}}$, which is the output of a multiplication or a squaring, a coefficient reduction algorithm should output a vector $\overline{\mathbf{w}}$ such that $\overline{\mathbf{w}} \sim \overline{\mathbf{z}}$, in the sense of (3.1), whose components are approximately the same size as t. As observed in [14], the equivalence relation (3.1) is captured by an underlying lattice, and finding $\overline{\mathbf{w}}$ is tantamount to solving an instance of the *closest vector problem* (CVP) in this lattice. To see why this is, we first fix some notation as in [14].

Let $\overline{\mathbf{u}}$ and $\overline{\mathbf{v}}$ be vectors in $\mathbb{Z}^{m+1}$ such that the following condition is satisfied:

$$[t^m, \ldots, t, 1] \cdot \overline{\mathbf{u}}^T \equiv [t^m, \ldots, t, 1] \cdot \overline{\mathbf{v}}^T \pmod{t^{m+1}-1}$$

Then we say that $\overline{\mathbf{u}}$ *is congruent to* $\overline{\mathbf{v}}$ *modulo* $t^{m+1}-1$ and write this as $\overline{\mathbf{u}} \cong_{t^{m+1}-1} \overline{\mathbf{v}}$. Note that this is exactly the same as saying $\psi^{-1}(\overline{\mathbf{u}}) \equiv \psi^{-1}(\overline{\mathbf{v}}) \pmod{t^{m+1}-1}$, and so $\overline{\mathbf{u}} \sim \overline{\mathbf{v}} \iff \overline{\mathbf{u}} \cong_{t^{m+1}-1} \overline{\mathbf{v}}$.

Similarly, but abusing notation slightly, for any integer $b \neq t^{m+1}-1$ (where b is typically a power of the word base of the target architecture), we write $\overline{\mathbf{u}} \cong_b v$ for some integer

v satisfying $[t^m, \ldots, t, 1] \cdot \overline{\mathbf{u}}^T \equiv v \pmod{b}$, and say $\overline{\mathbf{u}}$ *is congruent to* v *modulo* b, in this case. We reserve the use of '$\equiv$' to express a component-wise congruence relation, i.e., $\overline{\mathbf{u}} \equiv \overline{\mathbf{v}} \pmod{b}$. Finally, we denote by $\overline{\mathbf{u}} \bmod b$ the component-wise modular reduction of $\overline{\mathbf{u}}$ by b.

The lattice underlying the equivalence relation (3.1) can now enter the frame. Let $\mathbf{V} = \{\overline{\mathbf{v}}_0, \ldots, \overline{\mathbf{v}}_m\}$ be a set of $m+1$ linearly independent vectors in $\mathbb{Z}^{m+1}$ such that $\overline{\mathbf{v}}_i \cong_{t^{m+1}-1} \overline{\mathbf{0}}$, the all zero vector, for $i = 0, \ldots, m$. Then the set of all integer combinations of elements of $\mathbf{V}$ forms an integral lattice, $\mathcal{L}(\mathbf{V})$, with the property that for all $\overline{\mathbf{z}} \in \mathbb{Z}^{m+1}$, and all $\overline{\mathbf{u}} \in \mathcal{L}$, we have

$$\overline{\mathbf{z}} + \overline{\mathbf{u}} \cong_{t^{m+1}-1} \overline{\mathbf{z}} \tag{3.2}$$

In particular, the equivalence relation (3.1) is captured by the lattice $\mathcal{L}$, in the sense that

$$\overline{\mathbf{x}} \cong_{t^{m+1}-1} \overline{\mathbf{y}} \iff \overline{\mathbf{x}} - \overline{\mathbf{y}} \in \mathcal{L}$$

Therefore if one selects basis vectors for $\mathcal{L}$ that have infinity-norm approximately t, then for a given $\overline{\mathbf{z}} \in \mathbb{Z}^{m+1}$, finding the closest vector $\overline{\mathbf{u}} \in \mathcal{L}$ to $\overline{\mathbf{z}}$ (with respect to the infinity-norm), means the vector $\overline{\mathbf{w}} = \overline{\mathbf{z}} - \overline{\mathbf{u}}$ is in the fundamental domain of $\mathcal{L}$, and so has components of the desired size. Furthermore, since $\overline{\mathbf{w}} = \overline{\mathbf{z}} - \overline{\mathbf{u}}$, by (3.2) we have

$$\overline{\mathbf{w}} \cong_{t^{m+1}-1} \overline{\mathbf{z}},$$

and hence solving the CVP in this lattice solves the CRP. In general solving the CVP is NP-hard, but since we can exhibit a good (near-othogonal) lattice basis for LWPFIs, and an excellent lattice basis for GRPs, solving it is straightforward in our case.

3.4 Lattice basis and simple reduction

For GRPs, we use the following basis for $\mathcal{L}$:

$$\begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & -t \\ -t & 1 & \cdots & 0 & 0 & 0 \\ 0 & -t & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & -t & 1 & 0 \\ 0 & 0 & \cdots & 0 & -t & 1 \end{bmatrix} \tag{3.3}$$

Observe that the infinity-norm of each basis vector is t, so elements in the fundamental domain will have components of the desired size, and that each basis vector is orthogonal to all others except the two adjacent vectors (considered cyclically). In order to perform a simple reduction that reduces the size of components by approximately $\log_2 t$ bits, write each component of $\overline{\mathbf{z}}$ in base t: $z_i = z_{i,1}t + z_{i,0}$. If we define $\overline{\mathbf{w}}^T$ to be:

$$\begin{bmatrix} z_m \\ z_{m-1} \\ \vdots \\ \vdots \\ z_1 \\ z_0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & -t \\ -t & 1 & \cdots & 0 & 0 & 0 \\ 0 & -t & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & -t & 1 & 0 \\ 0 & 0 & \cdots & 0 & -t & 1 \end{bmatrix} \begin{bmatrix} z_{m-1,1} \\ z_{m-2,1} \\ \vdots \\ \vdots \\ z_{0,1} \\ z_{m,1} \end{bmatrix},$$

then $\overline{\mathbf{w}} \cong_{t^{m+1}-1} \overline{\mathbf{z}}$ and each $|w_i| \approx |z_i|/t$, assuming $|z_i| > t^2$. This was the method of reduction described in [12], which requires integer division. The idea described in [13] was based on an analogue of Barrett reduction [2]. The method we shall use, from [14], is based on Montgomery reduction [42] and for t not a power of 2 is the most efficient of the three Chung-Hasan methods.

3.5 Montgomery lattice-basis reduction

In ordinary Montgomery reduction [42], one has an integer $0 \leq Z < pR$ which is to be reduced modulo p, an odd prime, where here R is the smallest power of the word base b larger than p. The central idea is to add a multiple of p to Z such that the result is divisible by R. Upon dividing by R, which is a simple right shift of words, the result is congruent to $ZR^{-1} \pmod{p}$, and importantly is less than $2p$.

In the context of GRPs, let $R = b^q$ be the smallest power of b greater than t. The input to the reduction algorithm is a vector $\overline{\mathbf{z}} \in \mathbb{Z}^{m+1}$ for which each component is approximately R^2. The natural analogue of Montgomery reduction is to add to $\overline{\mathbf{z}}$ a vector $\overline{\mathbf{u}} \in \mathcal{L}$ whose components are also bounded by R^2, such that $\overline{\mathbf{z}} + \overline{\mathbf{u}} \equiv [0, \ldots, 0] \pmod{R}$. Then upon the division of each component by R, the result will be a vector $\overline{\mathbf{w}}$ which satisfies

$$\overline{\mathbf{w}} \cong_{t^{m+1}-1} (\overline{\mathbf{z}} + \overline{\mathbf{u}}) \cdot R^{-1} \cong_{t^{m+1}-1} \overline{\mathbf{z}} \cdot R^{-1},$$

and which has components of the desired size. While this introduces an R^{-1} term into the congruence, as with Montgomery arithmetic, one circumvents this simply by altering the original coset representation of $\mathbb{Z}/(t^{m+1}-1)\mathbb{Z}$, via the map $x \mapsto xR \pmod{t^{m+1}-1}$, which is bijective since $\gcd(t^{m+1}-1, R) = 1$, assuming t is even, see §5. How then does one find a suitable lattice point $\overline{\mathbf{u}}$? For this one use the lattice basis (3.3), which from here on in we call L. Proposition 3 of [14] proves that $\det L = 1 - t^{m+1}$, and so $\gcd(\det L, R) = 1$. One can therefore compute

$$\overline{\mathbf{u}}^T \stackrel{\text{def}}{=} -L^{-1} \cdot \overline{\mathbf{z}}^T \pmod{R}, \tag{3.4}$$

$$\overline{\mathbf{w}}^T \stackrel{\text{def}}{=} (\overline{\mathbf{z}}^T + L \cdot \overline{\mathbf{u}}^T)/R, \tag{3.5}$$

giving $\overline{\mathbf{w}}$ with the required properties. Observe that the form of these two operations is identical to Montgomery reduction, the only difference being that integer multiplication is replaced by matrix by vector multiplication. It is easy to see that this is what one requires, since for any $\overline{\mathbf{u}} \in \mathbb{Z}^{m+1}$, we have $L \cdot \overline{\mathbf{u}}^T \in \mathcal{L}$, and so

$$\overline{\mathbf{z}}^T + L \cdot \overline{\mathbf{u}}^T \cong_{t^{m+1}-1} \overline{\mathbf{z}}^T.$$

Furthermore, modulo R we have

$$\overline{\mathbf{z}}^T + L \cdot \overline{\mathbf{u}}^T = \overline{\mathbf{z}}^T + L \cdot (-L^{-1} \cdot \overline{\mathbf{z}}^T \bmod R) \equiv [0, \ldots, 0]^T,$$

ensuring the division of each component by R is exact. Hence $\overline{\mathbf{w}} \cong_{t^{m+1}-1} \overline{\mathbf{z}} \cdot R^{-1}$, as claimed.

In [14], an algorithm was given for computing $\overline{\mathbf{u}}$ and $\overline{\mathbf{w}}$ in (3.4) and (3.5) respectively, for an arbitrary LWPFI $f(t)$. The number of word-by-word multiply instructions in the algorithm — which is the dominant cost — is $\approx nq^2$, where n is the degree of $f(t)$, and $R = b^q$. In comparison, for ordinary Montgomery reduction modulo an integer of equivalent size this number is n^2q^2, making the former approach potentially very attractive. For our choice of primes — the GRPs — our specialisation of this algorithm is extremely efficient, as we show in §5.

3.6 High level view of Chung-Hasan arithmetic

For extension fields, there exists a natural separation between the polynomial arithmetic of the extension, and the prime subfield arithmetic, which makes respective optimisation considerations for each almost orthogonal. On the other hand, if for an LWPFI one naively attempts to use efficient techniques that are valid for extension fields, then one encounters an inherent obstruction, namely that there is no such separation between the polynomial arithmetic and the coefficient arithmetic, which leads to coefficient expansion upon performing arithmetic operations. Chung-Hasan arithmetic can be viewed as a tool to overcome this obstruction, since it provides an efficient solution to the coefficent reduction problem. In practice therefore any efficient techniques for extension field arithmetic can be ported to prime fields, whenever the prime is an LWPFI, which is precisely what we do in §4.

4 GRP Multiplication

In this section we detail algorithms for performing multiplication of GRP residue representatives. While for the reduction and residue representation we consider elements to be in $\mathbb{Z}^{m+1}$, the multiplication algorithm arises from the arithmetic of the polynomial ring $\mathbb{Z}[T]/(T^{m+1}-1)\mathbb{Z}[T]$, and so here we use this ring to derive the multiplication formulae.

4.1 Ordinary multiplication formulae

Let $\mathcal{R} = \mathbb{Z}[T]/(T^{m+1}-1)\mathbb{Z}[T]$, and let $\overline{\mathbf{x}} = [x_m, \ldots, x_0]$ and $\overline{\mathbf{y}} = [y_m, \ldots, y_0]$ be elements in $\mathcal{R}$. Then in $\mathcal{R}$ the product $\overline{\mathbf{x}} \cdot \overline{\mathbf{y}}$ is equal to $[z_m, \ldots, z_0]$, where

$$z_i = \sum_{j=0}^{m} x_{\langle j \rangle} y_{\langle i-j \rangle}, \tag{4.1}$$

where the subscript $\langle i \rangle$ denotes $i \pmod{m+1}$. This follows from the trivial property $T^{m+1} \equiv 1 \pmod{T^{m+1}-1}$, and that for $\overline{\mathbf{x}} = \sum_{i=0}^{m} x_i T^i$ and $\overline{\mathbf{y}} = \sum_{j=0}^{m} y_j T^j$, we have:

$$\begin{aligned}\overline{\mathbf{x}} \cdot \overline{\mathbf{y}} &= \sum_{i=0}^{m} x_i \cdot (T^i \cdot \overline{\mathbf{y}}) = \sum_{i=0}^{m} x_i \cdot \Big(\sum_{j=0}^{m} y_j T^{i+j} \Big) \\ &= \sum_{i=0}^{m} x_i \cdot \Big(\sum_{j=0}^{m} y_{\langle j-i \rangle} T^j \Big) = \sum_{j=0}^{m} \Big(\sum_{i=0}^{m} x_i \cdot y_{\langle j-i \rangle} \Big) T^j.\end{aligned}$$

This is of course just the cyclic convolution of $\overline{\mathbf{x}}$ and $\overline{\mathbf{y}}$.

4.2 Multiplication formulae of Nogami *et al.*

Nogami, Saito and Morikawa proposed the use of all-one polynomials (AOPs) to define extensions of prime fields [44]. In this section we will first describe their algorithm in this context, and then show how it fits into the framework developed in §3.

Let $\mathbb{F}_p$ be a prime field and let $f(\omega) = \omega^m + \omega^{m-1} + \cdots + \omega + 1$ be irreducible over $\mathbb{F}_p$, i.e., $m+1$ is prime and p is a primitive root modulo $m+1$. Then $\mathbb{F}_{p^m} = \mathbb{F}_p[\omega]/(f(\omega)\mathbb{F}_p[\omega])$. Using

the polynomial basis $\{\omega^m, \omega^{m-1}, \ldots, \omega\}$ — rather than the more conventional $\{\omega^{m-1}, \ldots, \omega, 1\}$ — elements of $\mathbb{F}_{p^m}$ are represented as vectors of length m over $\mathbb{F}_p$:

$$\overline{\mathbf{x}} = [x_m, \ldots, x_1] = x_m\omega^m + x_{m-1}\omega^{m-1} + \cdots + x_1\omega.$$

Let $\overline{\mathbf{x}} = [x_m, \ldots, x_1]$ and $\overline{\mathbf{y}} = [y_m, \ldots, y_1]$ be two elements to be multiplied. For $0 \leq i \leq m$, let

$$q_i = \sum_{j=1}^{m/2} (x_{\langle \frac{i}{2}+j \rangle} - x_{\langle \frac{i}{2}-j \rangle})(y_{\langle \frac{i}{2}+j \rangle} - y_{\langle \frac{i}{2}-j \rangle}), \tag{4.2}$$

where the subscript $\langle i \rangle$ here, as in §4.1, denotes $i \pmod{m+1}$. One then has:

$$\overline{\mathbf{z}} = \overline{\mathbf{x}} \cdot \overline{\mathbf{y}} = \sum_{i=1}^{m} z_i\omega^i, \quad \text{with } z_i = q_0 - q_i. \tag{4.3}$$

Nogami *et al.* refer to these coefficient formulae as the *cyclic vector multiplication algorithm* (CVMA) formulae. The CVMA formulae are remarkable, since the number of $\mathbb{F}_p$ multiplications is reduced relative to the schoolbook method from m^2 to $m(m+1)/2$, but at the cost of increasing the number of $\mathbb{F}_p$ additions from $m^2 - 1$ to $3m(m-1)/2 - 1$. As alluded to in §3.6, a basic insight of the present work is the observation that one may apply the expressions in (4.2) to GRP multiplication, provided that one uses the Chung-Hasan representation and reduction methodology of §3, to give a full modular multiplication algorithm.

Note that Karatsuba-Ofman multiplication [30] offers a similar trade-off for extension field arithmetic. Crucially however, as we show in §4.6, when we apply these formulae to GRPs the number of additions required is in fact reduced. One thus expects the CVMA to be significantly more efficient at contemporary ECC bitlengths. The original proof of (4.3) given in [44] excludes some intermediate steps and so for the sake of clarity we give a full proof in §4.4, beginning with the following motivation.

4.3 Alternative bases

Observe that in the set of equations (4.2), each of the $2(m+1)$ coefficients x_j, y_j is featured $m+1$ times, and so there is a nice symmetry and balance to the formulae. However due to the choice of basis, both x_0 and y_0 are implicitly assumed to be zero. The output $\overline{\mathbf{z}}$ naturally has this property also, and indeed if one extends the multiplication algorithm to compute z_0 we see that it equals $q_0 - q_0 = 0$.

At first sight, the expression $z_i = q_0 - q_i$ may seem a little unnatural. It is easy to change the basis from $\{\omega^m, \ldots, \omega\}$ to $\{\omega^{m-1}, \ldots, \omega, 1\}$: for $\overline{\mathbf{x}} = [x_{m-1}, \ldots, x_0]$ and $\overline{\mathbf{y}} = [y_{m-1}, \ldots, y_0]$, we have:

$$\overline{\mathbf{z}} = \overline{\mathbf{x}} \cdot \overline{\mathbf{y}} = \sum_{i=0}^{m-1} z_i\omega^i,$$

resulting in the expressions $z_i = q_m - q_i$, with q_i as given before. This change of basis relies on the relation

$$\omega^m \equiv -1 - \omega - \cdots - \omega^{m-1} \bmod f(\omega). \tag{4.4}$$

Note that in using this basis we have implicitly ensured that $x_m = y_m = 0$ in (4.2), rather than $x_0 = y_0 = 0$, and again the above formula is consistent since $z_m = q_m - q_m = 0$. More generally if one excludes ω^k from the basis, then $x_k = y_k = 0$ and $z_i = q_k - q_i$.

One may infer from these observations that the most natural choice of basis would seem to be $\{\omega^m, \ldots, \omega, 1\}$, and that the expressions for q_i arise from the arithmetic in the quotient ring $\mathcal{R}' = \mathbb{F}_p[\omega]/((\omega^{m+1} - 1)\mathbb{F}_p[\omega])$, rather than $\mathbb{F}_{p^m} = \mathbb{F}_p[\omega]/(f(\omega)\mathbb{F}_p[\omega])$. In this case multiplication becomes

$$\overline{\mathbf{z}} = \overline{\mathbf{x}} \cdot \overline{\mathbf{y}} = \sum_{i=0}^{m-1} z_i \omega^i = \sum_{i=0}^{m-1} (q_m - q_i)\omega^i = \sum_{i=0}^{m} -q_i \omega^i,$$

where for the last equality we have again used equation (4.4).

4.4 Derivation of coefficient formulae

We now derive the CVMA formulae of (4.2). Let $\overline{\mathbf{x}} = [x_m, \ldots, x_0] = \sum_{i=0}^{m} x_i \omega^i$, and $\overline{\mathbf{y}} = [y_m, \ldots, y_0] = \sum_{i=0}^{m} y_i \omega^i$. Then in the ring $\mathcal{R}'$, as in (4.1) the product $\overline{\mathbf{x}} \cdot \overline{\mathbf{y}}$ is equal to $\sum_{i=0}^{m} z_i \omega^i$, where

$$z_i = \sum_{j=0}^{m} x_{\langle j \rangle} y_{\langle i-j \rangle}.$$

Of crucial importance is the following identity. For $0 \leq i \leq m$ we have:

$$2\sum_{j=0}^{m} x_{\langle j \rangle} y_{\langle i-j \rangle} - 2\sum_{j=0}^{m} x_{\langle j \rangle} y_{\langle j \rangle} = -\sum_{j=0}^{m} (x_{\langle j \rangle} - x_{\langle i-j \rangle})(y_{\langle j \rangle} - y_{\langle i-j \rangle}). \tag{4.5}$$

To verify this identity observe that when one expands the terms in the right-hand side, the two negative sums cancel with the second term on the left-hand side, since both are over a complete set of residues modulo $m + 1$. Similarly the two positive sums are equal and therefore cancel with the convolutions in the first term on the left-hand side. We now observe that there is some redundancy in the right-hand side of (4.5), in the following sense. First, observe that

$$\sum_{j=0}^{m} x_{\langle \frac{i}{2}+j \rangle} y_{\langle \frac{i}{2}-j \rangle} = \sum_{j=0}^{m} x_{\langle \frac{i}{2}+(j-\frac{i}{2}) \rangle} y_{\langle \frac{i}{2}-(j-\frac{i}{2}) \rangle} = \sum_{j=0}^{m} x_{\langle j \rangle} y_{\langle i-j \rangle}.$$

One can therefore rewrite the right-hand side of (4.5) as:

$$-\sum_{j=0}^{m} (x_{\langle \frac{i}{2}+j \rangle} - x_{\langle \frac{i}{2}-j \rangle})(y_{\langle \frac{i}{2}+j \rangle} - y_{\langle \frac{i}{2}-j \rangle}). \tag{4.6}$$

Noting that the $j = 0$ term of expression (4.6) is zero, we rewrite it as:

$$-\sum_{j=1}^{m/2} (x_{\langle \frac{i}{2}+j \rangle} - x_{\langle \frac{i}{2}-j \rangle})(y_{\langle \frac{i}{2}+j \rangle} - y_{\langle \frac{i}{2}-j \rangle}) - \sum_{j=m/2+1}^{m} (x_{\langle \frac{i}{2}+j \rangle} - x_{\langle \frac{i}{2}-j \rangle})(y_{\langle \frac{i}{2}+j \rangle} - y_{\langle \frac{i}{2}-j \rangle}),$$

which in turn becomes

$$-\sum_{j=1}^{m/2} (x_{\langle \frac{i}{2}+j \rangle} - x_{\langle \frac{i}{2}-j \rangle})(y_{\langle \frac{i}{2}+j \rangle} - y_{\langle \frac{i}{2}-j \rangle}) - \sum_{j=1}^{m/2} (x_{\langle \frac{i}{2}-j \rangle} - x_{\langle \frac{i}{2}+j \rangle})(y_{\langle \frac{i}{2}-j \rangle} - y_{\langle \frac{i}{2}+j \rangle}),$$

and then upon negating the two terms in the second summation, we finally have

$$-\sum_{j=0}^{m}(x_{\langle\frac{i}{2}+j\rangle}-x_{\langle\frac{i}{2}-j\rangle})(y_{\langle\frac{i}{2}+j\rangle}-y_{\langle\frac{i}{2}-j\rangle})=2\sum_{j=1}^{m/2}(x_{\langle\frac{i}{2}+j\rangle}-x_{\langle\frac{i}{2}-j\rangle})(y_{\langle\frac{i}{2}+j\rangle}-y_{\langle\frac{i}{2}-j\rangle}).$$

Hence (4.5) becomes

$$\sum_{j=0}^{m}x_{\langle j\rangle}y_{\langle i-j\rangle}=\sum_{j=0}^{m}x_{\langle j\rangle}y_{\langle j\rangle}-\sum_{j=1}^{m/2}(x_{\langle\frac{i}{2}+j\rangle}-x_{\langle\frac{i}{2}-j\rangle})(y_{\langle\frac{i}{2}+j\rangle}-y_{\langle\frac{i}{2}-j\rangle}). \tag{4.7}$$

Equation (4.7) gives an expression for the coefficients of the product $\overline{\mathbf{z}}$ of elements $\overline{\mathbf{x}}$ and $\overline{\mathbf{y}}$, in the ring $\mathcal{R}'$. Assuming these are computed using the more efficient right-hand side, in order to restrict back to $\mathbb{F}_p[\omega]/(f(\omega)\mathbb{F}_p[\omega])$, one can reduce the resulting polynomial $\overline{\mathbf{z}}$ by $f(\omega)$. Note however that one does not need to use a smaller basis à la Nogami *et al.* in §4.2 or §4.3 , but can reduce by $f(\omega)$ *implicitly*, without performing any computation. Indeed, letting $\langle\overline{\mathbf{x}},\overline{\mathbf{y}}\rangle=\sum_{j=0}^{m}x_{\langle j\rangle}y_{\langle j\rangle}$, we have:

$$\begin{aligned}\overline{\mathbf{z}}&=\sum_{i=0}^{m}z_i\omega^i=\sum_{i=0}^{m}(-q_i+\langle\overline{\mathbf{x}},\overline{\mathbf{y}}\rangle)\omega^i=\sum_{i=0}^{m}-q_i\omega^i+\langle\overline{\mathbf{x}},\overline{\mathbf{y}}\rangle\sum_{i=0}^{m}\omega^i\\&\equiv\sum_{i=0}^{m}-q_i\omega^i\pmod{f(\omega)}.\end{aligned} \tag{4.8}$$

Therefore the first term on the right-hand side of (4.7) vanishes, so that one need not even compute it. Thus using the arithmetic in $\mathcal{R}'$ but implicitly working modulo $f(\omega)$ is more efficient than performing arithmetic in $\mathcal{R}'$ alone. This is somewhat fortuitous as it means that while the multiply operation in (4.8) is not correct in $\mathcal{R}'$, nevertheless, when one maps back to $\mathbb{F}_p[\omega]/(f(\omega)\mathbb{F}_p[\omega])$, it is correct.

4.5 Application to GRPs

Since equation (4.5) is an algebraic identity, it is easy to see that exactly the same argument applies in the context of GRPs, and we can replace the formulae (4.1) with the CVMA formulae (4.2). Since reduction in the ring $\mathcal{R}=\mathbb{Z}[T]/(T^{m+1}-1)\mathbb{Z}[T]$ has a particularly nice form for GRPs, we choose to use the full basis for $\mathcal{R}$ and hence do not reduce *explicitly* modulo $\Phi_{m+1}(T)$ to obtain a smaller basis. This also has the effect of eliminating the need to perform the addition of q_0 (or q_m, or whichever term one wants to eliminate when one reduces modulo $\Phi_{m+1}(T)$), simplifying the multiplication algorithm further. Absorbing the minus sign into the q_i, Algorithm 1 details how to multiply residue representatives.

Remark 1. Observe that each component of $\overline{\mathbf{z}}$ may be computed entirely independently of the others. Hence using $m+1$ processors rather than 1, it would be possible to speed up the execution time of Algorithm 1 by a factor of $m+1$, making it particularly suitable for hardware implementation. In §5 we consider the parallelisation of our reduction algorithms as well.

ALGORITHM 1: **GRP MULTIPLICATION**

INPUT: $\overline{\mathbf{x}} = [x_m, \ldots, x_0], \overline{\mathbf{y}} = [y_m, \ldots, y_0] \in \mathbb{Z}^{m+1}$
OUTPUT: $\overline{\mathbf{z}} = [z_m, \ldots, z_0] \in \mathbb{Z}^{m+1}$
where $\overline{\mathbf{z}} \cong_{\Phi_{m+1}(t)} \overline{\mathbf{x}} \cdot \overline{\mathbf{y}}$

1. For $i = m$ to 0 do:
2. $\quad z_i \leftarrow \sum_{j=1}^{m/2} (x_{\langle \frac{i}{2} - j \rangle} - x_{\langle \frac{i}{2} + j \rangle}) \cdot (y_{\langle \frac{i}{2} + j \rangle} - y_{\langle \frac{i}{2} - j \rangle})$
3. Return $\overline{\mathbf{z}}$

4.6 Cost comparison

We here use a simple cost model to provide a measure of the potential performance improvement achieved by using Algorithm 1, rather than schoolbook multiplication of residues. We assume the inputs to the multiplication algorithm have coefficients bounded by b^q, i.e., they each consist of q words. Let $M(q, q)$ be the cost of a q-word by q-word schoolbook multiplication, and let $A(q, q)$ be the cost of an ition of two q-word values. We assume that $A(2q, 2q) = 2A(q, q)$ and that there is no overflow beyond $2q$ words in the resulting vector components, which one can ensure by selecting appropriate GRPs, see §7. The cost of the multiplication using each method is as follows.

GRP schoolbook multiplication Working modulo $T^m + \cdots + T + 1$ and using a basis consisting of m terms only, the number of coefficient multiplications is m^2, while the number of double-length additions is also m^2. Hence the total cost is simply

$$m^2 \cdot M(q, q) + 2m^2 \cdot A(q, q).$$

Note that computing the convolution (4.1) costs

$$(m + 1)^2 \cdot M(q, q) + 2m(m + 1) \cdot A(q, q),$$

which is costlier since it requires embedding into $\mathcal{R}$, which introduces some redundancy.

CVMA formulae For each z_i computing each term in the sum costs $M(q, q) + 2A(q, q)$, and so computing all these terms costs $\frac{m}{2} \cdot (M(q, q) + 2A(q, q))$. The cost of adding these is $(\frac{m}{2} - 1)A(2q, 2q) = (m - 2) \cdot A(q, q)$. For all the $m + 1$ terms z_i the total cost is therefore

$$\frac{m(m + 1)}{2} \cdot M(q, q) + 2(m^2 - 1) \cdot A(q, q).$$

Therefore by using the CVMA formulae, we reduce not only the number of multiplications, but also the number of additions (by 2), contrary to the case of field extensions, for which the CVMA formulae increases the number of additions by nearly 50%. We have thus found an analogue of the asymptotic cyclic versus linear convolution speed up at small bitlengths for which schoolbook multiplication is optimal, for GRPs.

5 GRP Reduction

In this section we detail reduction algorithms for two types of GRPs. The first, Algorithm 2, assumes only that t is even, which provides the minimum possible restriction on the form of the resulting GRPs for any given bitlength. All such GRPs can therefore be implemented with code parametrised by the single variable t, which may be beneficial for some applications. Supposing that $R = b^q > t$, then as with Montgomery reduction, it is more efficient to reduce components not by R as in (3.4) and (3.5), but by b sequentially q times. In Algorithm 2 each reduction therefore reduces the input's components by approximately $\log_2 b$ bits.

The second reduction method as detailed in Algorithm 3 is a specialisation of Algorithm 2. It assumes that $t \equiv 0 \bmod 2^l$ for some $l > 1$, and each application of the reduction function reduces the input's components by approximately l bits. Algorithm 3 is potentially far more efficient than Algorithm 2, depending on the form of t. Ideally one should choose a t for which $l > (\log_2 t)/2$ so that two applications of the reduction function are sufficient in order to produce components of the desired size, which is minimal. In general for other values of l a larger number of reductions may be needed, which we consider in §7. In constrast to Algorithm 2, which is designed for generality, Algorithm 3 is geared towards high-speed reduction. The trade-off arising here is that there will naturally be far fewer GRPs of this restricted form. We also present a modification of Algorithm 3, which is slightly more efficient in practice, in Algorithm 4.

5.1 GRP reduction: t even

Following §3.5, in equation (3.4) we need the matrix $-L^{-1}$:

$$-L^{-1} = \frac{1}{t^{m+1}-1} \begin{bmatrix} 1 & t^m & \cdots & t^3 & t^2 & t \\ t & 1 & \cdots & t^4 & t^3 & t^2 \\ t^2 & t & \cdots & t^5 & t^4 & t^3 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ t^{m-1} & t^{m-2} & \cdots & t & 1 & t^m \\ t^m & t^{m-1} & \cdots & t^2 & t & 1 \end{bmatrix}. \tag{5.1}$$

The form of L and $-L^{-1}$ allows one to compute $\overline{\mathbf{u}} = -L^{-1} \cdot \overline{\mathbf{z}} \bmod b$ and $L \cdot \overline{\mathbf{u}}$, computed in equation (3.5), very efficiently. Since t is even, the following vector may be computed. Let $t[0]$ be the least significant digit of t, written in base b, and let

$$\overline{\mathbf{V}} \stackrel{\text{def}}{=} \frac{1}{t[0]^{m+1}-1}[t[0]^m, t[0]^{m-1}, \ldots, t[0], 1] \bmod b.$$

Algorithm 2 details how to reduce a given an input vector $\overline{\mathbf{z}}$ by b, modulo $t^{m+1}-1$, given the precomputed vector $\overline{\mathbf{V}}$. Observe that Algorithm 2 greatly simplifies the reduction algorithm originally given in [14]. This is possible since for $t^{m+1}-1$ one can interleave the computation of the vectors $\overline{\mathbf{u}}$ and $\overline{\mathbf{w}}$ defined in (3.4) and (3.5) respectively. This has two benefits. First, as one computes each component of $\overline{\mathbf{w}}$ sequentially, one need only store a single component of $\overline{\mathbf{u}}$, rather than $m+1$. Second, since when one computes $L \cdot \overline{\mathbf{u}}$ one needs to compute $t \cdot u_{\langle i+1 \rangle}$ for $i = m, \ldots, 0$ (in `line 3`), one obtains $t[0] \cdot u_i$ (the first term on right-hand side of `line 4`) for free by computing the full product $t \cdot u_{\langle i+1 \rangle}$ first. One therefore avoids recomputing the least significant digit of $t \cdot u_{\langle i+1 \rangle}$ in each loop iteration. In fact one can do this for any polynomial

$t^{m+1} - c$, with exactly the same algorithm, the only difference being in the definition of $\overline{\mathbf{V}}$, where $t^{m+1} - c$ becomes the denominator. For polynomials with other non-zero coefficients, this does not seem possible, and so Algorithm 2 seems likely to be the most efficient Chung-Hasan reduction possible with this minimal restriction on the form of t.

ALGORITHM 2: $\mathrm{red1}_b(\overline{\mathbf{z}})$

`INPUT:` $\overline{\mathbf{z}} = [z_m, \ldots, z_0] \in \mathbb{Z}^{m+1}$
`OUTPUT:` $\mathrm{red}_b(\overline{\mathbf{z}})$ `where` $\mathrm{red}_b(\overline{\mathbf{z}}) \cong_{t^{m+1}-1} \overline{\mathbf{z}} \cdot b^{-1}$

1. `Set` $u_0 \leftarrow (\sum_{i=0}^{m} V_i \cdot z_i[0]) \bmod b$
2. `For` $i = m$ `to` 0 `do:`
3. $\quad v_i \leftarrow t \cdot u_{\langle i+1 \rangle}$
4. $\quad u_i \leftarrow (v_i[0] - z_i[0]) \bmod b$
5. $\quad w_i \leftarrow (z_i + u_i - v_i)/b$
6. `Return` $\overline{\mathbf{w}}$

It is straightforward to verify that Algorithm 2 correctly produces an output vector in the correct congruency class, via a sequence of simple transformations of [14, Algorithm 3]. However we do not do so here, since we are mainly interested in the more efficient Algorithms 3 and 4.

Remark 2. Note that in the final loop iteration, u_0 from `line 1` is recomputed, which is therefore unnecessary. However, we chose to write the algorithm in this form to emphasise its cyclic structure. Indeed, there is no need to compute u_0 first; if one cyclically rotates $\overline{\mathbf{V}}$ by j places to the left, then the vector $\overline{\mathbf{w}}$ to be added to $\overline{\mathbf{z}}$ in (3.5) is rotated j places to the left also. One can therefore compute each coefficient of $\mathrm{red1}_b(\overline{\mathbf{z}})$ independently of the others using a rotated definition for $\overline{\mathbf{V}}$ (or equivalently by rotating the input $\overline{\mathbf{z}}$). This demonstrates that a parallelised version of the reduction algorithm with $m + 1$ processors is feasible. However, as each processor requires the least significant word of each component of $\overline{\mathbf{z}}$, this necessitates a synchronised broadcast before each invocation of the reduction function. In this scenario the reduction time would be proportional to the number of such broadcasts and reductions required, independently of $m + 1$.

5.2 GRP reduction: $t \equiv 0 \bmod 2^l$

In the ideal case that $t = 2^l$, we see that such a GRP would be a GMN. In this case, one can use the reduction method detailed in §3.4 without resorting to using its Montgomery version at all. Multiplication would also be faster thanks to Nogami's formulae. Unfortunately, such GRPs seem to be very rare. It is easy to show that if $t = 2^l$ with $l > 1$ and $\Phi_{m+1}(t)$ is prime, then $l = m + 1$. Testing the first few cases, we find prime GRPs for $l = 2, 3, 7, 59$ but no others for prime $l < 400$. Note that these primes contradict Dubner's assertion that no such GRPs exist [17, §2]. Since for $l = 59$ the corresponding GRP has 3422 bits, this is already out of our target range for ECC, so we need not worry about such GRPs.

Hoping not to cause confusion, in this subsection we now let $b = 2^l$ where l is not necessarily and usually not the word size of the target architecture. We denote the cofactor of b in t by c (which by the above discussion we assume is > 1), so that $t = b \cdot c$. Algorithm 3 details how to reduce a given an input vector $\overline{\mathbf{z}}$ by b, modulo $t^{m+1} - 1$.

ALGORITHM 3: $\mathrm{red2}_b(\overline{\mathbf{z}})$

INPUT: $\overline{\mathbf{z}} = [z_m, \ldots, z_0] \in \mathbb{Z}^{m+1}$
OUTPUT: $\mathrm{red}_b(\overline{\mathbf{z}})$ where $\mathrm{red}_b(\overline{\mathbf{z}}) \cong_{t^{m+1}-1} \overline{\mathbf{z}} \cdot b^{-1}$

1. For $i = m$ to 0 do:
2. $\quad w_i \leftarrow (z_i + (-z_i \bmod b))/b - c \cdot (-z_{\langle i+1 \rangle} \bmod b)$
3. Return $\overline{\mathbf{w}}$

A simple proof of correctness of Algorithm 3 comes from the specialisation of Algorithm 2. Since $t \equiv 0 \bmod b$, writing t in base b, the vector $\overline{\mathbf{V}}$ becomes

$$\overline{\mathbf{V}} \stackrel{\text{def}}{=} [0, \ldots, 0, -1] \bmod b.$$

Hence for line 1 of Algorithm 2 we have

$$u_0 \leftarrow -z_0[0] \bmod b.$$

Since in line 4 of Algorithm 2, we have $v_i \equiv 0 \bmod b$, we deduce that $u_i = -z_i \bmod b$, and hence we can eliminate u_i altogether. Each loop iteration then simplifies to

$$\begin{aligned} v_i &\leftarrow t \cdot (-z_{\langle i+1 \rangle} \bmod b) \\ w_i &\leftarrow (z_i + (-z_i \bmod b) - v_i)/b \end{aligned} \tag{5.2}$$

Upon expanding (5.2), we obtain

$$\begin{aligned} w_i &\leftarrow (z_i + (-z_i \bmod b))/b - t \cdot (-z_{\langle i+1 \rangle} \bmod b)/b \\ &= (z_i + (-z_i \bmod b))/b - c \cdot (-z_{\langle i+1 \rangle} \bmod b), \end{aligned}$$

as required. However since we did not provide a proof of correctness of Algorithm 2, we also give a direct proof as follows. Observe that modulo $t^{m+1} - 1$, we have

$$\begin{aligned} \psi^{-1}(\overline{\mathbf{w}}) &\equiv \sum_{i=0}^{m} w_i t^i \\ &\equiv \sum_{i=0}^{m} [(z_i + (-z_i \bmod b))/b - c \cdot (-z_{\langle i+1 \rangle} \bmod b)] t^i \\ &\equiv \sum_{i=0}^{m} (z_i/b) t^i + \sum_{i=0}^{m} (-z_i \bmod b))/b) t^i - \sum_{i=0}^{m} ((-z_{\langle i+1 \rangle} \bmod b)/b) t^{i+1} \\ &\equiv \sum_{i=0}^{m} z_i t^i/b \pmod{t^{m+1} - 1} \end{aligned}$$

as required. In terms of operations that may be performed very efficiently, we alter Algorithm 3 slightly to give Algorithm 4, which has virtually the same proof of correctness as the one just given.

ALGORITHM 4: $\mathrm{red3}_b(\overline{\mathbf{z}})$

INPUT: $\overline{\mathbf{z}} = [z_m, \ldots, z_0] \in \mathbb{Z}^{m+1}$
OUTPUT: $\mathrm{red}_b(\overline{\mathbf{z}})$ where $\mathrm{red}_b(\overline{\mathbf{z}}) \cong_{t^{m+1}-1} \overline{\mathbf{z}} \cdot b^{-1}$

1. For $i = m$ to 0 do:
2. $\quad w_i \leftarrow z_i/b + c \cdot (z_{\langle i+1 \rangle} \bmod b)$
3. Return $\overline{\mathbf{w}}$

Note that the first term in line 2 of Algorithm 3 has been replaced by a division by b, which can be effected as a simple shift, while now the second term needs the positive residue modulo b, which can be extracted more efficiently. Hence Algorithm 4 is the one we use. By our previous discussion, c necessarily has Hamming weight at least two for GRPs in our desired range. By using c that have very low Hamming weight, one can effect the multiplication by c by shifts and adds, rather than a multiply (or `imulq`) instruction. Hence for such GRPs, assuming only two invocations of Algorithm 4 are needed, reduction will be extremely efficient.

Remark 3. Regarding parallelisation, observe that for $m + 1$ processors, only the least significant word of $z_{\langle i+1 \rangle}$ is passed to processor i, thus reducing the broadcast requirement in comparison with Algorithm 2.

6 GRP Residue Representation

So far in our treatment of both multiplication and reduction, for the sake of generality we have assumed arbitrary precision when representing GRP residues in $\mathbb{Z}^{m+1}$. In this section we specialise to fixed precision and develop a residue representation that ensures that our chosen algorithms are efficient. Our decisions are informed purely by our chosen multiplication and reduction algorithms — Algorithms 1 and 4 — which we believe offer the best performance for GRPs for the relatively small bitlengths which are relevant to ECC. In other scenarios or if considering asymptotic performance, one would need to redesign the residue representation and multiplication algorithm accordingly.

For $x \in \{0, \ldots, t^{m+1} - 1\}$ we write $\overline{\mathbf{x}} = [x_m, \ldots, x_0]$ for its base-t expansion, i.e., $x = \sum_{i=0}^{m} x_i t^i$. The base-$t$ representation has positive coefficients, however Algorithm 1 makes use of negative coefficients, so we prefer to incorporate these. We therefore replace the mod function in the conversion with mods, the least absolute residue function, to obtain a residue in the interval $[-t/2, t/2 - 1]$:

$$\mathrm{mods}(x) = \begin{cases} x \bmod t & \text{if } (x \bmod t) < t/2, \\ x \bmod t - t & \text{otherwise.} \end{cases}$$

Using this function, Algorithm 5 converts residues modulo $t^{m+1} - 1$ into the required form [14, Algorithm 1].

ALGORITHM 5: **BASE-t CONVERSION** ψ

`INPUT:` An integer $0 \leq x < t^{m+1} - 1$
`OUTPUT:` $\overline{\mathbf{x}} = [x_m \ldots, x_0]$ such that $|x_i| \leq t/2$
and $\sum_{i=0}^{m} x_i t^i \equiv x \pmod{t^{m+1} - 1}$

1. For i from 0 to m do:
2. $\quad x_i \leftarrow x \bmod t$
3. $\quad x \leftarrow (x - x_i)/t$
4. $x_0 \leftarrow x_0 + x$
5. Return $\overline{\mathbf{x}} = [x_m, \ldots, x_0]$

The reason for `line 4` in Algorithm 5 is to reduce modulo $t^{m+1} - 1$ the coefficient of t^{m+1} possibly arising in the expansion. Note that in this addition, $x \in \{0, 1\}$, and hence $|x_i| \leq t/2$ for each $0 \leq i \leq m$. By construction, we in fact have $-t/2 \leq x_i < t/2$ for $1 < i < m$ while only x_0 can attain the upper bound of $t/2$. There are therefore $t^m(t+1)$ representatives in this format, thus introducing a very small additional redundancy. Letting $k = \lceil \log_2 t \rceil$, if we assume $t \leq 2^k - 2$, so that $[-t/2, t/2] \subset [-2^k/2, 2^k/2 - 1]$, then the coefficients as computed above can be represented in two's complement in k bits. In terms of efficiency, Algorithm 5 contains divisions by t, which requires not only time, but also space, which on some platforms may be at a premium. Writing $t = 2^l \cdot c$ as in §5.2, then if the cofactor $c = 2^{k-l} - c'$ with c' very small, then division by t consists of a shift right by l bits and a division by c, which can be performed efficiently using Algorithm 1 of [12].

Following this conversion, it might seem desirable to define vectors whose components are in $[-2^k/2, 2^k/2 - 1]$ to be reduced, or canonical residue representatives. However, for efficiency purposes it is preferable to have a reduction function which, when performed sufficiently many times, outputs an element for which one does not have to perform any modular additions or subtractions to make reduced, as this eliminates data-dependent branching. A control-flow invariant reduction function is also essential to defend against side-channel attacks, see §9. To obtain such a function, observe that the second term in `line 2` of Algorithm 4, namely $c \cdot (z_{\langle i+1 \rangle} \bmod b)$, is positive, and in the worst case is k bits long. The first term, z_i/b, is clearly $l = \log_2 b$ bits shorter than z_i. Since one adds these the resulting value may be $k + 1$ bits, or larger, depending on the initial length of the inputs' components. Furthermore, since we wish to allow negative components, in two's complement the output requires a further bit, giving a minimal requirement of $k + 2$ bits. We therefore choose not to use minimally reduced elements as coset representatives in $\mathbb{Z}^{m+1}/\sim$, as output by Algorithm 5, but slightly larger elements, which we now define.

Definition 5. *We define the following set of elements of* $\mathbb{Z}^{m+1}$ *to be* reduced*:*

$$\mathbb{I}^{m+1} = \{[x_m, \ldots, x_0] \in \mathbb{Z}^{m+1} \mid -2^{k+1} \leq x_i < 2^{k+1}\}. \tag{6.1}$$

Note that the redundancy inherent in this representation depends on how close t is to 2^{k+2}. For a modular multiplication, we assume that the inputs are reduced. We must therefore ensure that the output is reduced also. This naturally leads one to consider I/O stability, as we do in §7.

Once we have a reduced representative $\overline{\mathbf{x}} = \psi(x)$ we also need to convert to the Montgomery domain. While one can do this in $\mathbb{Z}/(t^{m+1}-1)\mathbb{Z}$ before applying ψ, it is more convenient to do so in $\mathbb{Z}^{m+1}/\sim$. Assuming q reductions by b are sufficient to ensure I/O modular multiplication stability, we precompute $\psi(b^{2q} \bmod \Phi_{m+1}(t))$ and then using Algorithms 1 and 4 compute

$$\overline{\mathbf{x}} \cdot \psi(b^{2q} \bmod \Phi_{m+1}(t))/b^q \cong_{\Phi_{m+1}(t)} \psi(x \cdot b^q).$$

Similarly, to get back from the Montgomery domain, again using Algorithms 1 and 4, we compute

$$\psi(x \cdot b^q) \cdot \psi(1)/b^q \cong_{\Phi_{m+1}(t)} \psi(x).$$

With regard to mapping back from $\overline{\mathbf{x}} = [x_m, \ldots, x_0] \in \mathbb{I}^{m+1}$ to canonical residues in $\mathbb{Z}/\Phi_{m+1}(t)\mathbb{Z}$, one has

$$\sum_{i=0}^{m} x_i t^i \equiv \sum_{i=0}^{m-1} (x_i - x_m) t^i \pmod{\Phi_{m+1}(t)},$$

which can be computed efficiently by first using Horner's rule and then mapped to $\{0, \ldots, \Phi_{m+1}(t)-1\}$ by repeated additions or subtractions. In terms of operations required for ECC, we assume that the conversions are one-time computations only, with all other operations taking place in the (Montgomery) Chung-Hasan representation.

7 Modular Multiplication Stability

In this section we analyse Algorithms 1 and 4 with a view to ensuring I/O stability for modular multiplication. We assume the following: $b = 2^l$, $t = c \cdot b$ where $c < 2^{k-l}$ (and hence $t < 2^k - 2$), and that reduced elements have the form (6.1). Input elements therefore have components in $\mathbb{I} = [-2^{k+1}, 2^{k+1} - 1]$, and these are representable in $k+2$ bits in two's complement. For simplicity and in order for our analysis to be as general as possible, we use the term single precision to mean a word base large enough to contain t — even if this in fact requires multiprecision on a given architecture — and double precision to mean twice this size. We assume that for this single precision word size w, the components of $\overline{\mathbf{z}}$ output by Algorithm 1 are double precision. In practice one prefers to specialise to actual single precision t on a given architecture, since this obviates the need for multiprecision arithmetic; utilising the native double precision multipliers that most CPUs possess is more efficient, and reduction is also faster for smaller t since fewer iterations need be performed. We note that in constrained environments however, multiprecision may however be unavoidable.

During the multiplication, terms of the form $x_i - x_j$ are computed, which are bounded by

$$-2^{k+2} + 1 \leq x_i - x_j \leq 2^{k+2} - 1,$$

and which therefore fit into $k+3$ bits in two's complement. The product of two such elements is performed, giving a result

$$-2^{2k+4} + 2^{k+3} - 1 \leq (x_i - x_j) \cdot (y_j - y_i) \leq 2^{2k+4} - 2^{k+3} + 1,$$

which fits into $2k+5$ bits in two's complement. One then adds $m/2$ of these terms, giving a possible expansion of up to $\lceil \log_2 m/2 \rceil$ bits, which must be double precision. We therefore have a constraint on the size of t (in addition to the constraint $t < 2^k - 2$) in terms of m:

$$\lceil \log_2 (m/2) \rceil + 2k + 5 \leq 2w \tag{7.1}$$

This inequality determines a constraint on the size of t, given m and w. Assuming (7.1) is satisfied, one then needs to find the minimum value of $b = 2^l$ such that the result of the multiplication step, when reduced by b a specified number of times, say q, outputs a reduced element. This needs to be done for each (m, k) found in the procedure above. Any power of 2 larger than this minimum will obviously be satisfactory also, however minimising b maximises the set of prime-producing cofactors c, which as stated in §5 may be useful in some scenarios.

In §6, we showed that one application of Algorithm 4 shortened an input's components by $l-1$ bits, unless the components were already shorter than $(k+2)+(l-1)$ bits. Therefore stipulating that q reductions suffice to produce a reduced output, we obtain a bound on l in the following manner. Let

$$h = \lceil \log_2 (m/2) \rceil + 2k + 5$$

Then after one reduction, the maximum length of a component is $h - l + 1$. Similarly after q reductions, the maximum length is $\max\{h - q(l-1), k+2\}$, and we need this to be at most $k + 2$. Hence our desired condition is

$$h - q(l-1) \leq k + 2$$

Solving for l, we have

$$l \geq 1 + \frac{\lceil \log_2 (m/2) \rceil + k + 3}{q} \tag{7.2}$$

Using these inequalities it is an easy matter to generate triples $(m+1, k, l)$ which ensure multiplication stability for any w and q. For example, for $w = 64$, Tables 1 and 2 give sets of stable parameters for $q = 2$ and $q = 3$ respectively.

Table 1. Stable parameters for $w = 64$, $q = 2$

$m+1$	k	l	$c <$	$\lceil \log_2 p \rceil$
3	61	33	2^{28}	122
5	61	34	2^{27}	244
7	60	34	2^{26}	360
11	60	34	2^{26}	600
13	60	34	2^{26}	720
17	60	34	2^{26}	960

Table 2. Stable parameters for $w = 64$, $q = 3$

$m+1$	k	l	$c <$	$\lceil \log_2 p \rceil$
3	61	23	2^{38}	122
5	61	23	2^{38}	244
7	60	23	2^{37}	360
11	60	23	2^{37}	600
13	60	23	2^{37}	720
17	60	23	2^{37}	960

The final column gives the maximum bitlength of a GRP that can be represented with those parameters, though of course by using smaller c one can opt for smaller primes, and the corresponding minimum value of l reduces according to (7.2). To generate suitable GRPs, a simple linear search over the values of c of the desired size is sufficient, checking whether or not $\Phi_{m+1}(2^l \cdot c)$ is prime, see §10.

8 Full GRP Modular Multiplication

For completeness we now piece together the parts treated thus far into a full modular multiplication algorithm, where in Algorithm 6 we assume q reductions by b are required for I/O stability and in `line 4` either Algorithm 2 or Algorithm 4 is used according to the form of b.

ALGORITHM 6: **GRP MODMUL**

`INPUT:` $\overline{\mathbf{x}} = [x_m, \ldots, x_0], \overline{\mathbf{y}} = [y_m, \ldots, y_0] \in \mathbb{I}^{m+1}$
`OUTPUT:` $\overline{\mathbf{z}} = [z_m, \ldots, z_0] \in \mathbb{I}^{m+1}$ `where` $\overline{\mathbf{z}} \cong_{\Phi_{m+1}(t)} \overline{\mathbf{x}} \cdot \overline{\mathbf{y}} \cdot b^{-q}$

1. `For` $i = m$ `to` 0 `do:`
2. $\quad z_i \leftarrow \sum_{j=1}^{m/2} (x_{\langle \frac{i}{2} - j \rangle} - x_{\langle \frac{i}{2} + j \rangle}) \cdot (y_{\langle \frac{i}{2} + j \rangle} - y_{\langle \frac{i}{2} - j \rangle})$
3. `For` k `from` 0 `to` $q - 1$ `do:`
4. $\quad \overline{\mathbf{z}} \leftarrow \mathrm{red}_b(\overline{\mathbf{z}})$
5. `Return` $\overline{\mathbf{z}}$

Should t be multiprecision on a particular architecture, then as with Montgomery arithmetic it may be more efficient to use an interleaved multiplication and reduction algorithm, as we detail in Algorithm 7. Here one needs b to be the word base of the underlying architecture and so in `line 6`, if $t \equiv 0 \pmod{b}$ we use Algorithm 4, otherwise we use Algorithm 2. For $\overline{\mathbf{x}} = [x_m, \ldots, x_0]$ we write $x_i = x_i[0] + x_i[1]b + \cdots + x_i[q-1]b^{q-1}$.

ALGORITHM 7: **GRP MODMUL (interleaved)**

`INPUT:` $\overline{\mathbf{x}} = [x_m, \ldots, x_0], \overline{\mathbf{y}} = [y_m, \ldots, y_0] \in \mathbb{I}^{m+1}$
`OUTPUT:` $\overline{\mathbf{z}} = [z_m, \ldots, z_0] \in \mathbb{I}^{m+1}$ `where` $\overline{\mathbf{z}} \cong_{\Phi_{m+1}(t)} \overline{\mathbf{x}} \cdot \overline{\mathbf{y}} \cdot b^{-q}$

1. $\overline{\mathbf{z}} \leftarrow [0, \ldots, 0]$
2. `For` $k = 0$ `to` $q - 1$ `do:`
3. $\quad$ `For` $i = m$ `to` 0 `do:`
4. $\quad\quad w_i \leftarrow \sum_{j=1}^{m/2} (x_{\langle \frac{i}{2} - j \rangle}[k] - x_{\langle \frac{i}{2} + j \rangle}[k]) \cdot (y_{\langle \frac{i}{2} + j \rangle} - y_{\langle \frac{i}{2} - j \rangle})$
5. $\quad \overline{\mathbf{z}} \leftarrow \overline{\mathbf{z}} + \overline{\mathbf{w}}$
6. $\quad \overline{\mathbf{z}} \leftarrow \mathtt{red}_b(\overline{\mathbf{z}})$
7. `Return` $\overline{\mathbf{z}}$

To verify the correctness of Algorithm 7, observe that for each of the $m+1$ components of $\overline{\mathbf{z}}$, after the last iteration of the outer loop we have:

$$\begin{aligned} z_i &= \sum_{j=1}^{m/2} \Big(\sum_{k=0}^{q-1} (x_{\langle \frac{i}{2}-j \rangle}[k] - x_{\langle \frac{i}{2}+j \rangle}[k])/b^{q-k} \Big) \cdot (y_{\langle \frac{i}{2}+j \rangle} - y_{\langle \frac{i}{2}-j \rangle}) \\ &= \sum_{j=1}^{m/2} ((x_{\langle \frac{i}{2}-j \rangle} - x_{\langle \frac{i}{2}+j \rangle})/b^q) \cdot (y_{\langle \frac{i}{2}+j \rangle} - y_{\langle \frac{i}{2}-j \rangle}). \end{aligned}$$

Hence when taken modulo $\Phi_{m+1}(t)$, we see that $\overline{\mathbf{z}}$ is congruent to:

$$\begin{aligned} \sum_{i=0}^{m} z_i \cdot t^i &\cong_{\Phi_{m+1}(t)} \sum_{i=0}^{m} \Big(\sum_{j=1}^{m/2} ((x_{\langle \frac{i}{2}-j \rangle} - x_{\langle \frac{i}{2}+j \rangle})/b^q) \cdot (y_{\langle \frac{i}{2}+j \rangle} - y_{\langle \frac{i}{2}-j \rangle}) \Big) \cdot t^i \\ &\cong_{\Phi_{m+1}(t)} \sum_{i=0}^{m} \Big(\sum_{j=1}^{m/2} (x_{\langle \frac{i}{2}-j \rangle} - x_{\langle \frac{i}{2}+j \rangle}) \cdot (y_{\langle \frac{i}{2}+j \rangle} - y_{\langle \frac{i}{2}-j \rangle}) \cdot t^i \Big) / b^q \\ &\cong_{\Phi_{m+1}(t)} \overline{\mathbf{x}} \cdot \overline{\mathbf{y}} \cdot b^{-q}, \end{aligned}$$

as required. As with ordinary Montgomery arithmetic, there are many possible ways to perform the interleaving, see [34] for example.

9 Other arithmetic and side-channel secure ECC

In addition to modular multiplication, one also needs to perform other arithmetic operations when implementing ECC point multiplication. In this section we detail how to perform these using our representation and briefly explain how it enables point multiplication to be made immune to various side-channel attacks.

9.1 Other arithmetic operations

Addition/subtraction To perform an addition or subtraction of two reduced elements $\overline{\mathbf{x}}, \overline{\mathbf{y}}$, we compute the following:

$$\overline{\mathbf{x}} \pm \overline{\mathbf{y}} = [x_m \pm y_m, \ldots, x_0 \pm y_0].$$

Note that the bounds on each of these components is $[-2^{k+2}, 2^{k+2} - 2]$, which are therefore not necessarily reduced. One could reduce the resulting element using the specialisation to GRPs of [14, Algorithm 5], which shows how to do this for a general LWPFI. Chung and Hasan refer to this process as *short coefficient reduction* (SCR), as opposed to full modular reduction. However, for ECC operations it is faster (and more secure) to simply ignore this expansion and rely on a later modular multiplication to perform the reduction, as is required when computing a point addition or doubling, see [5, 6] and §9.2.

Squaring When t is single precision, the CVMA formulae do not have any common subexpressions, as arises for ordinary integer residue squaring. In this case GRP squaring is performed using Algorithm 6. If t is multiprecision, then the components of a product $\overline{\mathbf{x}} \cdot \overline{\mathbf{y}}$ are computed as a sum of integer squares. In this case, one can eliminate common subexpressions to improve efficiency by nearly a factor of two (in the multiplication step). On the other hand, when using Algorithm 7 and its variants it may be difficult to eliminate common subexpressions efficiently [34].

Inversion and equality check Inversion seems difficult to perform efficiently in the GRP representation. If t were prime then it would be possible to use an analogue of the inversion/division algorithm of [45], exploiting the cyclicity $t^{m+1} \equiv 1 \pmod{t^{m+1} - 1}$. However, for our GRPs t is even and greater than 2. One can therefore opt to map back to $\mathbb{Z}/\Phi_{m+1}(t)\mathbb{Z}$, remaining in the Montgomery domain, and perform inversion using the binary extended Euclidean algorithm (see [32], for example) and modular multplying by the precomputed value $\psi(b^3 \bmod \Phi_{m+1}(t))$. Alternatively, for data-independent inversion, one can simply power by $\Phi_{m+1}(t) - 2$, as do the authors of [6]. Using projective coordinates can obviate the need for inversions altogether, however for many protocols inversion is unavoidable and when it is avoidable, in some scenarios such representations of points should be randomised after a point multiplication [43].

Since our representation possesses redundancy, equality checking is naturally problematic. We therefore opt to map back to $\mathbb{Z}/\Phi_{m+1}(t)\mathbb{Z}$ to check equality there — as for inversion — while remaining in the Montgomery domain. For ECC equality checking is usually a one-time computation per coordinate, and so again this operation does not greatly impinge upon efficiency.

9.2 Side-channel secure ECC

As we demonstrated in §7, by choosing t, l and $m + 1$ carefully, one can avoid the need to compute any final additions or subtractions when performing a modular reduction. This is an analogue to various results for ordinary Montgomery arithmetic [26, 55, 56]. The lack of a conditional addition/subtraction averts threats such as [48, 57], the latter of which applies directly to the NIST GMNs. Our modular multiplication algorithm is thus control-flow invariant with no data-dependent operations, making it immune to timing attacks [35] and simple power analysis (SPA).

In addition to making modular multiplications and squarings immune to timing attacks and SPA, one can also ensure that the computation of an entire elliptic curve point addition or doubling is also immune. To do so, one chooses a GRP with t divisible by a sufficiently high power of 2, so that during the course of an elliptic curve point operation, even if one ignores the coefficient expansion caused by additions/subtractions, these do not overflow and the modular reductions ensure the outputs are fully reduced elements. Note that this requires $b = 2^l \mid t$ to be a few bits longer than the minimum l-values listed in Tables 1 and 2: for reasons of space we do not include the analysis here. By doing so, a point addition or doubling becomes an atomic operation, where the sequence of arithmetic operations is entirely data-independent. In this case one only needs point-multiplication-level defences against timing attacks and SPA, such as the double-and-add-always algorithm due to Coron [15], or the use of Edwards curves, for which the addition formula can also be used for doubling [7]. Hence, ECC over GRPs may be straight-line coded, which is beneficial for both efficiency and security.

Lastly, our representation can also be made immune to differential power analysis (DPA) [36]. Observe that the embedding of $\mathbb{Z}/\Phi_{m+1}(t)\mathbb{Z}$ into $\mathbb{Z}/(t^{m+1} - 1)\mathbb{Z}$ can be randomised by adding to it a random multiple $r \cdot \Phi_{m+1}(t)$ for $r \in \{0, \ldots, (t-1)-1\}$. While our embedding is an example of 'operand scaling' [45, 54] which is used for faster reduction, the addition of a multiple of the modulus within a redundant scaled representation also acts as a countermeasure to DPA — such as Goubin's attack [25] on the randomised projective coordinates defence of Coron [15] — as shown by Smart, Oswald and Page [51]. In particular, the au-

thors show that this countermeasure thwarts DPA whenever the scaling factor is longer than the longest string of ones or zeros in the binary expansion of the initial modulus. For the NIST GMNs, this countermeasure requires a large scaling factor, making the defence inefficient and nullifying the benefits of using these moduli. For GRPs, the scaling factor is $t-1$, while the longest string of ones or zeros in the binary expansion of $\Phi_{m+1}(t)$ is $\lceil \log t \rceil - 1$. Since GRPs *already* use the larger ring, we acquire this defence for almost negligible cost. In particular the addition of a random multiple r of $\Phi_{m+1}(t)$ to an element $\overline{\mathbf{x}}$ has the form $[x_m + r, x_{m-1} + r, \ldots, x_0 + r]$, which only requires $m+1$ additions. Since DPA depends on the ability of an attacker to predict a specific bit in the representation of a given field element, if the representation of field elements is randomised in this way prior to every point multiplication, or even every modular multiplication, then DPA will not be feasible.

10 GRP Parameters

In this section we provide empirical data regarding the abundancy of fast-reduction GRPs at various bitlengths relevant to ECC. We also specify parameters that are optimal within this subfamily, which are therefore particularly suitable for efficient implementation.

10.1 Estimating the number of GRP parameters

As we saw from Tables 1 and 2, for a given prime $m+1$ and word size w, there is an upper bound on the length of a GRP that may be represented. Table 3 contains estimates (or exact counts) for the number of GRPs which are in accordance with the GRP field and residue representation set out in this work, for word size $w = 64$ and where $q = 2$ reductions using Algorithm 4 suffice to ensure I/O modular multiplication stability. Note that this reduction specification entails the greatest possible restriction on the available GRPs; using either larger q or Algorithm 2 means vastly more GRPs are availble. It is a simple matter to generate such GRPs, as follows; GRPs of any more general form can be found similarly.

For a desired GRP p of bitlength $\lceil \log p \rceil$, Table 1 gives the minimum value of prime $m+1$ which is adequate to represent GRPs of this size. The inequality (7.1) gives k_{max} which is the maximum bitlength of t that is representable, while (7.2) gives the minimum value l required in order for $t = 2^l \cdot c$ to be I/O stable. We estimate t_{max} simply as $2^{\lceil \log p \rceil / m}$, which implies a maximum value for c of $2^{\lceil \log p \rceil / m - l_{min}}$. Similarly for p of this precise bitlength, we estimate the minimum value of c as $2^{(\lceil \log p \rceil - 1)/m - l_{min}}$. We denote this interval by $I(c)$. To estimate $P(\text{prime})$, which is the probability that a given generalised repunit in our form is a GRP, we performed a linear search over $c \in I(c)$, counting the first $1,000$ primes and simply dividing by the length of the search. The estimated total number of GRPs satisfying our requirement that $q = 2$ is then given by $|I(c)| \cdot P(\text{prime})$.

For each of $m+1 = 5, 7$ and 11, Table 3 contains estimated counts for the largest GRPs representable. It also contains estimates (or exact counts) for the number of GRPs at the NIST GMN sizes $224, 256$ and 384. We also consider bitlength 512 rather than 521, since this conjecturally gives 256-bit security, with the larger prime $2^{521} - 1$ being nominated purely for fast reduction. Observe that the number of available GRPs for a given $m+1$ decreases as the size of p, and hence c decreases. The number available for bitlengths 383 and 384 is particularly low. However, should this be a concern for a particular application, one can see from Table 2 that by moving to GRPs for which 3 reductions suffices, $|I(c)|$ becomes much larger (3.71×10^5) and our estimate of the number of GRPs becomes over $5,000$. On the other

Table 3. Estimated GRP counts for $w = 64$, $q = 2$ and $t \equiv 0 \pmod{2^{l_{min}}}$

$\lceil \log p \rceil$	$m+1$	k_{max}	$\log t_{max}$	l_{min}	$\lvert I(c) \rvert$	$P(\text{prime})$	$\approx$ #GRPs
600	11	60	60.0	34	4.49×10^6	8.54×10^{-3}	38.4×10^3
599	11	60	59.9	34	4.19×10^6	9.05×10^{-3}	37.9×10^3
512	11	60	51.2	30	1.61×10^5	1.05×10^{-2}	1697
511	11	60	51.1	30	1.51×10^5	1.06×10^{-2}	1591
384	11	60	38.4	24	1448	9.67×10^{-3}	14
383	11	60	38.3	24	1352	1.33×10^{-2}	18
360	7	60	60.0	34	4.49×10^6	1.82×10^{-2}	81.7×10^3
359	7	60	59.9	34	4.19×10^6	1.77×10^{-2}	74.1×10^3
256	7	60	42.66	25	2.27×10^4	2.47×10^{-2}	561
255	7	60	42.5	25	2.02×10^4	2.63×10^{-2}	531
244	5	61	61.0	34	2.14×10^7	1.68×10^{-2}	3.58×10^5
243	5	61	60.75	34	1.80×10^7	1.72×10^{-2}	3.08×10^5
224	5	56	56.0	31	5.34×10^6	1.98×10^{-2}	1.06×10^5
223	5	56	55.75	31	4.49×10^6	1.88×10^{-2}	8.42×10^4

hand, since 384 is not too far beyond the upper bound for the size of GRP representable by $m + 1 = 7$, it may be preferable to trade 12-bits of security for much improved performance, see §11.

Table 4. Efficient-reduction GRPs for $w = 64$, $q = 2$ and $t \equiv 0 \pmod{2^{l_{min}}}$

$\lceil \log p \rceil$	GRP	S.C. Secure
511	$\Phi_{11}(2^{42} \cdot (2^9 + 1))$	Yes
381	$\Phi_{11}(2^{34} \cdot (2^4 - 1))$	Yes
380	$\Phi_{11}(2^{34} \cdot (2^4 + 1))$	Yes
270	$\Phi_7(2^{34} \cdot (2^{11} - 1))$	Yes
253	$\Phi_7(2^{27} \cdot (2^{15} + 1))$	Yes
253	$\Phi_7(2^{37} \cdot (2^5 + 1))$	Yes
243	$\Phi_5(2^{59} \cdot (2^2 - 1))$	No
228	$\Phi_5(2^{54} \cdot (2^3 - 1))$	Yes
224	$\Phi_5(2^{33} \cdot (2^{23} - 1))$	No
220	$\Phi_5(2^{52} \cdot (2^3 - 1))$	Yes

10.2 Hamming weight 2 parameters

As we showed in §5.2, there are no suitable GRPs in the ECC range for which $t = 2^l$. Hence the next best type of GRP parameter t will have Hamming weight equal to 2, where we allow c to have the form $2^{c'} + 1$ as well as $2^{c'} - 1$ when there is sufficient slack in the representation, since subtractions cost the same as additions. We list these GRPs in Table 4. The final column indicates whether or not the given GRP allows for atomic side-channel secure point additions and doublings, as per §9.2. Note that for $m + 1 = 5$ and $w = 64$ we can not represent GRPs any larger than 244-bits, and are thus short of the conjectured 128-bit ECC security level of 256-bits. One can therefore either move up to $m + 1 = 7$, which can represent GRPs

of up to 360-bits, or one can opt to reduce security by a few bits, for better performance. Indeed, in recent work Käsper argues that the NIST GMN prime P-224 $= 2^{224} - 2^{96} + 1$ offers a satisfactory trade-off between security and efficiency [31], when used as the basis of the elliptic curve Diffie-Hellman (ECDH) key exchange in the Transport Layer Security (TLS) protocol [28]. Bernstein has also implemented arithmetic mod P-224 [4]. Yet another possibilty at this security level are the GFN primes $\Phi_8(2^{41} \cdot (2^{15} - 1))$ and $\Phi_8(2^{50} \cdot (2^6 - 1))$, both of which have bitlength 224, but experiments with such GFNs have not yet been carried out. Of course in hardware, one can tailor the word base in order to achieve 256-bit atomic elliptic curve point operations without any residual slack.

11 Implementation and Results

In this section we provide details of our proof-of-concept implementation and results. For simplicity we consider field multiplication only, as this tends to be the bottleneck in ECC point multiplication, and hence one can gain an accurate indication of performance in this simple way.

In terms of performance, implementations can be compared using the eBATS benchmarking project [8]. For example, nearly all of the fastest 256-bit ECDH implementations in the literature [5,6,22,23,27,40] feature in eBATS. However, as it is difficult to get a fair comparison between our implementation of multiplication and the above ECDH implementations, we opt to compare our multiplication performance with that featured in the $\texttt{mp}\mathbb{F}_q$ benchmarking system due to Gaudry and Thomé [23], which allows for such a comparison. This has been ported to OS-X 10.5.8 with minor changes and executed on a platform using an Intel Core 2 Duo at 2.2Ghz. As stated in [6], to date $\texttt{mp}\mathbb{F}_q$ gives only the fourth fastest implementation of ECDH, based on Bernstein's `curve25519`, which utilises a non-standard representation of residues mod $2^{255} - 19$ and exploits the floating-point unit of specific instruction-set architectures to great effect. Nevertheless, by comparing the basic multiplication cost on the target architecture, one can obtain a crude estimate of the relative performance of our arithmetic with that of `curve25519`, and hence in turn with the other implementations featured in eBATS.

Our implementation consists of essentially two inline assembly operations targeted at the Core 2 processor. One accumulates the innermost sum of `line 2` of Algorithm 6, while the other performs a single instance of the reduction operation in `line 2` of Algorithm 4. There are two versions of the latter which correspond to whether or not the cofactor c is general — which hence requires an `imulq` instruction — or is specialised, i.e., with Hamming weight 2. These operations use the 64-bit operations available on the Core 2 and the extended register set available in x86-64, each using a mere 4 of the 15 registers available. This allows one to rely on normal C code to arrange these macros, and to handle data-storage. As a result the gcc compiler can generate all of the intermediate memory access instructions and schedule the usage of the other 11 registers available. This means that the same code can be reused for any field supported by Algorithm 6 — the only changes required are the parameter definitions. To generate a particular instance of the family of algorithms we use a simple wrapper written in Python that arranges the sequence of these operations required for the particular parameter choice of $m + 1$ and t.

To emphasise the relative simplicity of our implementation, we use only 64-bit scalar operations on the processor, and allow the compiler to schedule most of the output instructions. As a result we reach a throughput of slightly less than one operation per cycle. In comparison

the $\texttt{mp}\mathbb{F}_q$ implementation of `curve25519` uses SSE2 to reach a throughput of almost two operations per cycle (the theoretical maximum on the architecture). Although our implementation is less efficient (because we have spent less programmer time on the machine-dependent optimisation) the performance achieved is still higher. Scheduling a lower-level implementation on the processor would be an interesting challenge.

As explained in §5 and §10, within the reduction algorithm we have a trade-off between the number of GRPs available and performance. If one opts for a generic value of c then many GRPs are available, but the reduction involves a full `imulq` instruction with relatively high latency. If we specialise our choice of c to those with Hamming weight 2 then we can replace this instruction with a shift and an `add` or `sub` instruction to improve performance. We have measured the performance of both implementations. To ensure a fair comparison we have merged our code into $\texttt{mp}\mathbb{F}_q$ so that all algorithms are being tested with the same timing code. This timer executes 10^6 operations in the field, measuring the elapsed time. The reported figures are the mean execution time for the operation. Table 5 contains cycle counts for Montgomery arithmetic at various bitlengths, as well as the `curve25519` modular multiplication cycle count. Table 6 contains our results for GRP modular multiplication.

Table 5. $\text{mp}\mathbb{F}_q$ cycle counts for curve25519 and Montgomery arithmetic

Algorithm	Size (bits)	ModMul (cycles)
Mont.	64	30
Mont.	128	105
Mont.	192	195
`curve25519`	255	140
Mont.	256	280
Mont.	320	407
Mont.	384	563
Mont.	448	757
Mont.	512	981

Table 6. Cycle counts for GRP arithmetic

Parameters	Max size (bits)	ModMul (cycles)
$m+1$=5, $HW(c)$=2	244	96
$m+1$=5, general c	244	112
$m+1$=7, $HW(c)=2$	360	165
$m+1$=7, general c	360	182
$m+1$=11, general c	600	340

As shown in Table 1 and considered in §10.2, the closest size of field to `curve25519` that we can implement using $m+1=5$ is only 244-bits. This small reduction in field size is compensated by an increase in performance, requiring only 80% of the `curve25519` cycles per multiplication. Using the specialised reduction function for the 243-bit GRP $\Phi_5(2^{59}\cdot(2^2-1))$, this figure improves to 69%. Since the results for the first line of Table 6 apply also to

Hamming weight 2 GRPs smaller than 243-bits, we obtain the same modular multiplication performance for the 228-bit GRP $\Phi_5(2^{54} \cdot (2^3 - 1))$, while utilising the acquired slack in the representation to ensure atomic point doublings/additions as per §9.2. At 512-bits with general c, compared to Montgomery multiplication, GRP multiplication costs only 35% as many cycles. For GRPs of 600-bits, this proportion would naturally be even smaller, however at this size Karatsuba-Ofman multiplication may be faster than schoolbook multiplication. We thus expect that point multiplications using 224-bit and 512-bit GRPs to be competitive with the state-of-the-art in the literature, once optimised. In particular, by comparing our arithmetic with the modular multiplication used in [6], which is the benchmark for point multiplication at the 128-bit security level, one gains an idea of the potential performance of arithmetic mod $\Phi_5(2^{54} \cdot (2^3 - 1))$, for example. In [6], residues are also represented by five 64-bit words. Residue multiplication requires 25 `mul` instructions, as well as 4 `imul`, 20 `add` and 20 `adc` instructions. In comparison, to multiply $\overline{\mathbf{x}}$ and $\overline{\mathbf{y}}$ in our representation, the CVMA formulae are as follows:

$$
\begin{aligned}
z_0 &= (x_4 - x_1)(y_1 - y_4) + (x_3 - x_2)(y_2 - y_3),\\
z_1 &= (x_2 - x_4)(y_4 - y_2) + (x_1 - x_0)(y_0 - y_1),\\
z_2 &= (x_0 - x_2)(y_2 - y_0) + (x_4 - x_3)(y_3 - y_4),\\
z_3 &= (x_3 - x_0)(y_0 - y_3) + (x_2 - x_1)(y_1 - y_2),\\
z_4 &= (x_1 - x_3)(y_3 - y_1) + (x_0 - x_4)(y_4 - y_0),
\end{aligned}
$$

requiring only 10 `mul`, 20 `sub` and 5 `add` and 5 `adc` instructions. Since the respective reduction algorithms are quite similar with both requiring two rounds of shifts, masks and additions, one expects the GRP modular multiplication to be considerably faster, when optimised. However, since this paper is predominantly expositional, we leave such optimisations as open research.

12 Conclusion

We have proposed efficient algorithms for performing arithmetic modulo a large family of primes, namely the generalised repunit primes. The algorithms are simple to implement, are fast, are easily parallelisable, can be made side-channel secure, and all across a wide range of field sizes. The central contribution of this work is the development of the necessary theory, covering field and residue representation, as well as algorithms for performing efficient multiplication and reduction in these fields. We have also presented proof-of-concept implementation results which provide an empirical comparison with results in the literature, which we ensured is fair by using the `mp`$\mathbb{F}_q$ benchmarking procedure. Against Montgomery arithmetic we show an approximate 3-fold improvement in performance, and expect optimised implementations of point multiplications using our proposed family to be competitive with state-of-the-art implementations. We have thus presented a compelling argument in favour of a new approach to the secure and efficient implementation of ECC.

Acknowledgements

The authors would like to thank Dan Page for making several very useful comments and suggestions.

References

1. A.O.L. Atkin and F. Morain. Elliptic curves and primality proving. Math. Comp., 61(203):29-68, 1993.
2. P. Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor, In *Advances in CryptologyCRYPTO 86* Springer-Verlag, LNCS 263, 311323, 1987.
3. P.T. Bateman and R.A. Horn. A Heuristic Asymptotic Formula Concerning the Distribution of Prime Numbers. In *Math. Comp.* 16, pp. 363–367, 1962.
4. D.J. Bernstein. A software implementation of NIST P-224. Presentation at the 5th Workshop on Elliptic Curve Cryptography (ECC 2001), University of Waterloo, October 29-31, 2001. Slides available from `http://cr.yp.to/talks.html`.
5. D.J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography - PKC 2006*, LNCS 3958, 207–228. Springer-Verlag, 2006.
6. D.J. Bernstein, N. Duif, T. Lange, P. Schwabe and B. Yang. High-speed high-security signatures. Cryptology ePrint Archive, Report 2011/368, 2011.
7. D.J. Bernstein and T. Lange. Faster addition and doubling on elliptic curves. In *Advances in Cryptology ASIACRYPT 2007*, LNCS 4833, pp. 29–50, Springer-Verlag, 2007.
8. D.J. Bernstein and T. Lange (editors). eBACS: ECRYPT Benchmarking of Cryptographic Systems. `http://bench.cr.yp.to/ebats.html`.
9. I.F. Blake, R.M. Roth and G. Seroussi. Efficient Arithmetic in $GF(2^m)$ through Palindromic Representation. Technical Report HPL-98-134, 1998. Available from `http://www.hpl.hp.com/techreports/98/HPL-98-134.html`.
10. I.F. Blake, G. Seroussi, and N.P. Smart. Advances in Elliptic Curve Cryptography. *London Mathemtical Society Lecture Note Series*, 317, Cambridge University Press, 2005.
11. M. Brown, D. Hankerson, J. López, and A. Menezes Software Implementation of the NIST Elliptic Curves Over Prime Fields In *Topics in Cryptology CT-RSA 2001*, LNCS 2020, 250–265, Springer.
12. J. Chung A. Hasan. More Generalized Mersenne Numbers. In *Selected Areas in Cryptography*, volume 3006 of LNCS, 335 – 347. Springer, 2004.
13. J. Chung and A. Hasan. Low-Weight Polynomial Form Integers for Efficient Modular Multiplication. In *IEEE Trans. Comput.*, 56-1, 44–57, 2007.
14. J. Chung and A. Hasan. Montgomery Reduction Algorithm for Modular Multiplication Using Low-Weight Polynomial Form Integers. In *ARITH 18*, 230–239, 2007
15. J.S. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *Cryptographic Hardware and Embedded Systems CHES 99*, LNCS 1717, pp. 292–302, 1999.
16. R.E. Crandall, E.W. Mayer and J.S. Papadopoulos. The twenty-fourth fermat number is composite. In *Math. Comp.*, vol. 72, 243, pp. 1555–1572, 2003.
17. H. Dubner. Generalized Repunit Primes. In *Math. Comp.*, vol. 61, 204, pp. 927–930, 1993.
18. H. Dubner and Y. Gallot. Distribution of generalized Fermat prime numbers. In *Math. Comp.*, vol. 71, 238, pp. 825–832, 2002.
19. H. Dubner and T. Granlund. Primes of the Form $(b^n + 1)/(b + 1)$. In *Journal of Integer Sequences*, vol. 3, 2, Art. 0.0.2.7, 2000.
20. G. Drolet. A new representation of elements of finite fields $GF(2^m)$ yielding small complexity arithmetic circuits. *IEEE Trans. Comput.*, 47(9): 938–946, 1998.
21. FIPS 186-2. Digital Signature Standard. Federal Information Processing Standards Publication 186-2, US Department of Commerce/N.I.S.T. 2000.
22. S.D. Galbraith, X. Lin and M. Scott. Endomorphisms for Faster Elliptic Curve Cryptography on a Large Class of Curves. In *J. Cryptology*, vol. 24, no. 3, pp. 446–469, 2011.
23. P. Gaudry and E. Thomé. The mpFq library and implementing curve-based key exchanges. In *SPEED: Software Performance Enhancement for Encryption and Decryption*, ECRYPT Workshop, 49–64, 2007.
24. W. Geiselmann and D. Grollmann. VLSI design for exponentiation in $GF(2^m)$. *AUSCRYPT'90*, 398–405. Springer-Verlag, 2001.
25. L. Goubin. A refined power analysis attack on elliptic curve cryptosystems. In em Public Key Cryptography PKC 03, LNCS 2567, pp. 199–211, 2003.
26. G. Hachez and J.J. Quisquater. Montgomery Exponentiation with No Final Subtractions: Improved Results. In *Cryptographic Hardware and Embedded Systems (CHES)*, Springer-Verlag LNCS 1965, pp. 293–301, 2000.
27. Hüseyin Hisil. Elliptic curves, group law, and efficient computation. Ph.D. thesis, Queensland University of Technology, 2010. URL: `http://eprints.qut.edu.au/33233`.
28. Internet Engineering Task Force. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS), 2006. `http://www.ietf.org/rfc/rfc4492`.

29. T. Itoh and S. Tsuji. Structure of Parallel Multipliers for a Class of Fields $GF(2^m)$. In *Information and Computers*, vol. 8, 21–40, 1989.
30. A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. In *Soviet Physics*, Doklady 7, 595–596, 1963.
31. E. Käsper. Fast elliptic curve cryptography in OpenSSL. In *2nd Workshop on Real-Life Cryptographic Protocols and Standardization (RLCPS 2011)*, to appear, 2011.
32. D.E. Knuth. The Art of Computer Programming, **2** - Semi-numerical Algorithms. Addison-Wesley, 2nd edition, 1981.
33. N. Koblitz. Elliptic curve cryptosystems. In *Math. Comp.*, **48**, pp. 203–209, 1987.
34. C.K. Koç, T. Acar and B.S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. In *IEEE Micro*, vol. 16, 3, pp. 26–33, 1996.
35. P.C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems. In *Advances in Cryptology CRYPTO 96*, LNCS 1109, pp. 104–113, 1996.
36. P.C. Kocher, J. Jaffe and B. Jun. Differential power analysis. In *Advances in Cryptology CRYPTO 99*, LNCS 1666, pp. 388–397, 1999.
37. S. Kwon, C.H. Kim and C.P. Hong. Gauss Period, Sparse Polynomial, Redundant Basis, and Efficient Exponentiation for a Class of Finite Fields with Small Characteristic. In *ISAAC 2003*, LNCS 2906, pp. 736–745, 2003.
38. A.K. Lenstra and H.W. Lenstra. The Development of the Number Field Sieve. *LNM 1554*, Springer-Verlag, 1993.
39. H.W. Lenstra, Jr. Factoring integers with elliptic curves. Ann. of Math. (2), 126(3):649–673, 1987.
40. P. Longa and C.H. Gebotys. Efficient techniques for high-speed elliptic curve cryptography. In *Cryptographic hardware and embedded systems, CHES 2010*, LNCS 6225, pp. 80–94, Springer 2010.
41. V. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology - CRYPTO 85.* LNCS 218, Springer-Verlag, pp. 417–426, 1985.
42. P.L. Montgomery. Modular Multiplication without trial division. *Math. Comp.*, **44**, 519–521, 1985.
43. D. Naccache, N.P. Smart and J. Stern. Projective Coordinates Leak. In *Advances in Cryptology - EUROCRYPT*, LNCS 2586, pp. 257–267, Springer-Verlag, 2004.
44. Y. Nogami, A. Saito, and Y. Morikawa. Finite Extension Field with Modulus of All-One Polynomial and Representation of Its Elements for Fast Arithmetic Operations. In *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences* Vol.E86-A No.9, 2376–2387, 2003.
45. E. Ozturk, B. Sunar, and E. Savas. Low-Power Elliptic Curve Cryptography Using Scaled Modular Arithmetic. In CHES 2004, Springer Verlag, LNCS 3156, pp 92–106, 2004.
46. D.S. Phatak and T. Goff. Fast Modular Reduction for Large Wordlengths via One Linear and One Cyclic Convolution. In 17th IEEE Symposium on Computer Arithmetic (ARITH'05), pp. 179–186, 2005.
47. R.L. Rivest, Shamir A., and L.M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Comm. ACM*, **21**, 120 – 126, 1978.
48. Y. Sakai and K. Sakurai. Simple Power Analysis on Fast Modular Reduction with Generalized Mersenne Prime for Elliptic Curve Cryptosystems. In *Ieice Transactions - IEICE*, vol. 89-A, no. 1, pp. 231–237, 2006.
49. A. Schinzel and W. Sierpiński. Sur certaines hypothèses concernant les nombres premiers. In *Acta Arith.* 4 (1958), pp. 185–208, Erratum 5 (1959), 259.
50. J.H. Silverman. Fast Multiplication in Finite Fields $GF(2^N)$. In *Proc. Workshop Cryptographic Hardware and Embedded Systems (CHES '99)*, LNCS 1717, 122–134. Springer 1999.
51. N.P. Smart, E. Oswald and D. Page. Randomised representations. In *IET Information Security*, vol. 2, 2, pp. 19–27, 2008.
52. W.M. Snyder. Factoring repunits. In *Amer. Math. Monthly*, 89 pp. 462–466, 1982.
53. J.A. Solinas. Generalized Mersenne Numbers. Technical report CORR-39, Dept. of C&O, University of Waterloo, 1999. Available from `http://www.cacr.math.uwaterloo.cahttp://citeseer.ist.psu.edu/solinas99generalized.html`
54. C.D. Walter. Faster Modular Multiplication by Operand Scaling. *Advances in Cryptology* LNCS 576, 313–323, Springer Verlag, 1992.
55. C.D. Walter. Montgomery Exponentiation Needs No Final Subtractions. In *Electronics Letters*, **35**, pp. 1831–1832, 1999.
56. C.D. Walter. Montgomery's Multiplication Technique: How to Make it Smaller and Faster. In *Cryptographic Hardware and Embedded Systems (CHES)*, Springer-Verlag LNCS 1717, pp. 80–93, 1999.
57. C.D. Walter and S. Thompson. Distinguishing Exponent Digits by Observing Modular Subtractions. In *CT-RSA 2001*, LNCS 2020, pp. 192–207, 2001.

58. H.C. Williams and E. Seah. Some primes of the form $(a^n - 1)/(a - 1)$. In *Math. Comp.* 33, pp. 1337–1342, 1979.
59. J.K. Wolf. Low Complexity Finite Field Multiplication. In *Discrete Math.*, no.s 106/107, 497–502, 1992.
60. H. Wu, A. Hasan, I. Blake and S. Gao. Finite Field Multiplier Using Redundant Representation. *IEEE Trans. Comput.*, Vol 51, Num 11, Nov 2002.
61. S. Yekhanin. Towards 3-query locally decodable codes of subexponential length. STOC '07, Proceedings of the thirty-ninth annual ACM symposium on Theory of computing, 266–274, 2007.