High-Performance Scalar Multiplication Using 8-Dimensional GLV/GLS Decomposition

Joppe W. Bos[1], Craig Costello[1], Huseyin Hisil[2], and Kristin Lauter[1]

[1] Microsoft Research, Redmond, USA
[2] Yasar University, Izmir, Turkey

Abstract. This paper explores the potential for using genus 2 curves over quadratic extension fields in cryptography, motivated by the fact that they allow for an *8-dimensional* scalar decomposition when using a combination of the GLV/GLS algorithms. Besides lowering the number of doublings required in a scalar multiplication, this approach has the advantage of performing arithmetic operations in a 64-bit ground field, making it an attractive candidate for embedded devices. We found cryptographically secure genus 2 curves which, although susceptible to index calculus attacks, aim for the standardized 112-bit security level. Our implementation results on both high-end architectures (Ivy Bridge) and low-end ARM platforms (Cortex-A8) highlight the practical benefits of this approach.

1 Introduction

Elliptic curve cryptography [29,34] is a popular approach to realize public-key cryptography. One of the main reasons to employ elliptic curves, rather than using more traditional settings like finite fields, is efficiency. According to [41], the performance gain when transferring the Diffie-Hellman protocol [13] from finite fields to elliptic (genus 1) curves at the 128-bit security level is an order of magnitude. There is an active research area dedicated to enhancing the core operation in curve-based protocols: the scalar multiplication. A novel approach that facilitates fast scalar multiplications is the Gallant-Lambert-Vanstone (GLV) method [18]. If an elliptic curve $E(\mathbb{F}_q)$ comes equipped with a non-trivial endomorphism, then a scalar k can be decomposed into two "mini-scalars", both of which are approximately half the bit-length of k: merging these mini-scalars means that the number of required point doublings in the scalar multiplication can be reduced by a factor of two. The GLV method was extended by Galbraith, Lin and Scott (GLS) [17], who show that regardless of the existence of an endomorphism on $E(\mathbb{F}_q)$, one can achieve a decomposition by considering the points $E(\mathbb{F}_{q^m})$ for $m > 1$. Furthermore, [17] explains that if E already comes equipped with a useful endomorphism over $\mathbb{F}_q$, then the GLV and GLS endomorphisms can be combined to achieve higher degree decompositions and increased performance. At Asiacrypt 2012, Longa and Sica [33] demonstrated this GLV/GLS combination to achieve a 4-dimensional scalar decomposition on elliptic curves

G. Bertoni and J.-S. Coron (Eds.): CHES 2013, LNCS 8086, pp. 331–348, 2013.

over the quadratic extensions of a large prime field (i.e. $E(\mathbb{F}_{p^2})$), and set the current software speed record for computing scalar multiplications over non-binary fields. The authors of [8] recently showed the practical potential of hyperelliptic (genus 2) curves in cryptography. One attractive aspect of genus 2 curves is that, in general, their Jacobian group $\mathrm{Jac}_C(\mathbb{F}_p)$ has a larger endomorphism ring than that of genus 1 curves. This means that over prime fields or over extension fields of the same degree, the highest possible degree of the GLV/GLS decomposition is twice as large in genus 2 as it is in genus 1.

In this paper we consider 8-dimensional scalar decompositions by exploring the use of genus 2 curves over quadratic extension fields. To the best of our knowledge, this is the first time an 8-dimensional scalar decomposition has been implemented and studied in detail, addressing two of the open problems posed in the original GLS paper [17, §9]. Using decompositions of this size leads to practical performance issues that do not arise in the 2- and 4-dimensional case; we highlight some pitfalls and present solutions in a variety of scenarios. In contrast to elliptic curves, "faster-than-generic" attacks are known on genus 2 curves over $\mathbb{F}_{p^2}$. Namely, one can use the "Weil descent" attack [15] to map the discrete logarithm problem to a higher dimensional abelian variety over $\mathbb{F}_p$, where index calculus attacks are possible [2,19]. We assess the current state-of-the-art in index calculus attacks [21,12] to give conservative security estimates, which present a strong case for the curves we use at the currently standardized 112-bit security level [40].

Since most high-end hardware architectures work with 64-bit words and many embedded platforms work with 32-bit words (like the ARM), using 64-bit primes means that our arithmetic in the ground field is respectively performed using one and two computer words only. We explore different approaches for arithmetic in $\mathbb{F}_p$, while using lazy reduction techniques from the pairing community [3] to achieve efficient arithmetic in $\mathbb{F}_{p^2}$. In addition to the 8-dimensional GLV/GLS approach, we consider "generic" genus 2 curves (curves which do not exploit any special properties) and the Kummer surface over $\mathbb{F}_{p^2}$. Our implementation results on a 64-bit Ivy Bridge processor and a Cortex-A8 ARM CPU show that this approach is competitive with the current state-of-the-art in elliptic curve cryptography, although we reiterate that our work targets the 112-bit security level, while most of the work we (are able to) compare against targets the 128-bit security level. Our implementations targeting 64-bit platforms will be made publicly available through [6].

2 Preliminaries

In this paper we work with "imaginary" hyperelliptic curves of genus 2 over a quadratic extension of large prime fields. Such curves can be written as $C/\mathbb{F}_{p^2} : y^2 = x^5 + f_3x^3 + f_2x^2 + f_1x + f_0$. We use $\mathrm{Jac}_C(\mathbb{F}_{p^2})$ to denote the Jacobian group and we abbreviate the Mumford representation of general (i.e. weight 2) divisors on $\mathrm{Jac}_C(\mathbb{F}_{p^2})$ to write $(x^2 + u_1x + u_0, v_1x + v_0)$ as (u_1, u_0, v_1, v_0) in affine space, or as $(U_1 : U_0 : V_1 : V_0 : Z)$ in homogeneous projective space.

We explore three algorithms for computing scalar multiplications on $\mathrm{Jac}_C(\mathbb{F}_{p^2})$: (i) the generic algorithm which computes the scalar multiplication using a sequence of divisor doublings and additions only, (ii) the combination of the GLV [18] and GLS [17] algorithms which both exploit endomorphisms (but in a different way) to accelerate computations, and (iii) Gaudry's fast formulas [20] for arithmetic on a Kummer surface associated to $\mathrm{Jac}_C(\mathbb{F}_{p^2})$.

GLV and GLS Algorithms. The Gallant, Lambert and Vanstone (GLV) method [18] involves using special curves that come equipped with efficiently computable endomorphisms other than Frobenius. For example, when $p \equiv 1 \bmod \ell$ for an odd prime ℓ, the curve $C/\mathbb{F}_p : y^2 = x^\ell + a$ comes equipped with $\phi : (x, y) \mapsto (\xi_\ell x, y)$, for ξ_ℓ a non-trivial ℓ-th root of unity in $\mathbb{F}_p$. On the other hand, the Galbraith, Lin and Scott (GLS) method [17] does not rely on curves of a special form, but rather exploits the fact that, for any curve defined over $\mathbb{F}_p$, the p-power Frobenius endomorphism π_p acts non-trivially on points in extension fields of $\mathbb{F}_p$. Galbraith *et al.* [17, §3] further show how the GLV and GLS ideas can be combined to give more advantageous decompositions. Namely, for curves that are both defined over extension fields *and* have additional (non-trivial) endomorphisms, they show that this is achieved by taking the isogeny ϕ (constituting ψ) to be the twisting isomorphism corresponding to the additional endomorphism(s) on C. For special Buhler-Koblitz curves [10] of the form $C/\mathbb{F}_{p^2} : y^2 = x^5 + a$, we discuss this combined approach in detail in Section 4.

The Kummer Surface. Gaudry [20] showed that scalar multiplications can be computed more efficiently on a Kummer surface associated to the Jacobian of genus 2 curves than on the Jacobian itself. Recently, the authors of [8] used Gaudry's fast formulas on genus 2 curves over prime fields to set a new speed record for computing constant-time scalar multiplications. In this work we carry these techniques across to curves defined over quadratic extension fields, and since the method of using the Kummer surface essentially remains unchanged, we refer to [8, §5] for the details.

The CM Method over Quadratic Extension Fields. To obtain cryptographically strong genus 2 curves over $\mathbb{F}_{p^2}$, where p is a prime suitable for fast arithmetic as described in Section 5, we use the complex multiplication (CM) method. To find strong curves over $\mathbb{F}_{p^2}$ instead of over $\mathbb{F}_p$, we search for CM fields where p decomposes in a different way. The details are explained in [24], and we use the specific constructions in [24, §3.6.5, Ex. 5 and 6].

3 Curve Choices and Security

Weil Descent and Index Calculus. Attacks which are asymptotically "faster-than-generic" are known to exist on curves over extension fields, using a combination of the ideas of Weil descent and index calculus (see for example [15,2,19,22,16,11,23,21]). In this work we are concerned with the best-known attacks on the discrete logarithm problem (DLP) in the Jacobian of a genus 2

curve C defined over a quadratic extension field $\mathbb{F}_{p^2}$. Following [11,23], one attack transfers the DLP on $\mathrm{Jac}(C)(\mathbb{F}_{p^2})$ to the Jacobian of a higher genus curve $\tilde{C}$ which lies on the abelian variety over $\mathbb{F}_p$ obtained via Weil restriction of scalars from $\mathrm{Jac}(C)(\mathbb{F}_{p^2})$ [16, §7.1 - Ex. 7]. In general it can be hard to find such a curve $\tilde{C}$, and for the curves we use, the best known technique finds curves $\tilde{C}$ of genus 8 to use in the attack detailed in [11,23]. Certain cases of genus 2 (imaginary) hyperelliptic curves C over quadratic extension fields $\mathbb{F}_{p^2}$ have been classified as "weak" [45,35,26], in that their special form makes it easier than usual to find a suitable curve $\tilde{C}$ on the Weil restriction of $\mathrm{Jac}(C)$. None of the curves we use fall into these weak classifications: we can essentially rule this out by ensuring that our curves cannot be written as $C : y^2 = (x - \alpha) \cdot h(x)$, with $h(x) \in \mathbb{F}_p[x]$. Thus, to the best of our knowledge, the fastest attack on our curves is due to Gaudry [21], with further improvements provided by Nagao [39]. Gaudry's attack works directly on the abelian variety obtained as the Weil restriction of scalars, and solves the discrete logarithm problem on genus g hyperelliptic curves over $\mathbb{F}_{p^n}$, where both n and g are *fixed*, in heuristic asymptotic running time $\tilde{\mathcal{O}}(p^{2-\frac{2}{ng}})$, i.e. not including the "constants" depending on n and g and the logarithmic factors in p. For the sake of obtaining a better comparison with the generic Pollard rho algorithm, we reveal *some* of the factors that are hidden by the $\tilde{\mathcal{O}}$. One of the constants in the $\tilde{\mathcal{O}}$ depends exponentially on both g and n as $2^{3n(n-1)g}$ [39]. Hence, a conservative lower bound on the asymptotic running time of this attack, expressed in terms of group operations on the genus g curve, is $\mathcal{O}(p^{2-\frac{2}{ng}} \cdot 2^{3n(n-1)g} \cdot \log(p)^r)$ for some $r \geq 1$. To give a modest security estimate for our genus 2 curves over quadratic extension fields ($g = n = 2$), we take $r = 1$, ignore other constants involved and keep the $\mathcal{O}$ in terms of group operations on the dimension 4 abelian variety obtained as the Weil restriction of $\mathrm{Jac}(C)$. Hence, we arrive at $p^{3/2} \cdot 2^{12} \cdot \log(p)$ group operations as a conservative estimate of a lower bound on the complexity of Gaudry's attack for genus 2 curves over $\mathbb{F}_{p^2}$.

Generic Curves, Buhler-Koblitz Curves, and Kummer Surfaces. For each of the 3 algorithms (generic, Kummer, 8-GLV/GLS) considered in this work, we used the CM method to find curves over quadratic extension fields with characteristic less than 2^{64} that fall into 3 different categories: those which use a Montgomery-friendly prime of the form $(2^{31}-c_1)\cdot 2^{32}-1$ to target the 32-bit (ARM) environment, those which use a NIST-friendly prime of the form $2^{64}-c_2$ to target 64-bit platforms, and those which use the Mersenne prime $2^{61}-1$ that can employ specialized Montgomery- and NIST-like reduction (cf. Section 5). We note that all our fields[1] have $p \equiv 3 \bmod 4$, so that the quadratic extension can always be constructed as $\mathbb{F}_{p^2} = \mathbb{F}_p[i]/(i^2+1)$. Table 1 summarizes the curves that we use in this paper together with the arithmetic approach taken (Montgomery, NIST or special) and the security claims. The curve parameters are given in the

[1] We also considered the prime $p = 2^{64} - 2^{32} + 1 \equiv 1 \bmod 4$ which looks attractive for 32-bit platforms using NIST-like reduction, however our experiments showed that the Montgomery-friendly primes were faster.

Table 1. An overview of our implementations targeting the 112-bit security level. The security estimate (in bits) resulting from index calculus (i.c.) and Pollard rho (rho) attack are stated. For each instance, we state the prime p and the bit-lengths of the cofactor h and prime r where the group order is $h \cdot r$. For the Kummer instances, we also show the size of the prime (sub)group order r' of the twist.

algorithm	reduction	base field p	$\lvert h\rvert_2$	$\lvert r\rvert_2$	$\lvert r'\rvert_2$	security (bits) rho	security (bits) i.c.
generic	special	$2^{61}-1$	38	207	-	103	109
	Mont.	$(2^{31}-307656)\cdot 2^{32}-1$	36	217	-	108	112
	NIST	$2^{64}-189$	36	221	-	110	113
Kummer	special	$2^{61}-1$	38	207	228	103	109
	Mont.	$(2^{31}-307656)\cdot 2^{32}-1$	36	217	245	108	112
	NIST	$2^{64}-189$	36	221	250	110	113
8-GLV/GLS	special	$2^{61}-1$	32	213	-	105	109
	Mont.	$(2^{31}-201)\cdot 2^{32}-1$	31	222	-	109	112
	NIST	$2^{64}-2285$	33	224	-	111	113

full version of this paper [9]. The security estimate for the Pollard rho attack [43] is obtained using $\log_2\left(\sqrt{\frac{\pi r}{2\#\mathrm{Aut}}}\right)$, where #Aut is the size of the automorphism group of C. In our case all of the GLV/GLS curves have $\#\mathrm{Aut} = 10$, while all the other curves have $\#\mathrm{Aut} = 2$. The runtime of the index calculus attack depends on p, while the complexity of the Pollard rho attack depends on the (sub)group order r. When searching for curves, we aimed to balance the attack complexity of both approaches in order to enhance performance: relaxing the size of r does not decrease the level of claimed security for index-calculus, but results in smaller scalars (and faster scalar multiplications). This explains why the subgroup orders r in Table 1 are significantly smaller than 256 bits – our target for the Pollard rho security was to aim slightly below our estimate for the index calculus algorithms for the sake of being conservative. Of the 10 isomorphism classes of Buhler-Koblitz curves over $p = 2^{61}-1$, we chose the one corresponding to the Jacobian group with the largest prime factor of size 213 bits.

4 8-Dimensional GLV/GLS

4.1 8-GLV/GLS on Buhler-Koblitz Curves over $\mathbb{F}_{p^2}$

Following the description in [17, §5], we use a BK curve of the form $C/\mathbb{F}_{p^2} : y^2 = x^5 + u^{10}$, with $p \equiv 1 \bmod 10$ and $u^{10} \in \mathbb{F}_{p^2}$ such that $u \in \mathbb{F}_{p^{20}}$. Let $C'/\mathbb{F}_p : y^2 = x^5 + 1$. The map $\phi^{-1} : C \to C'$ defined as $\phi^{-1} : (x, y) \mapsto (x/u^2, y/u^5)$ takes points in $C(\mathbb{F}_{p^2})$ to points in $C'(\mathbb{F}_{p^{20}})$, where the p-power Frobenius map $\pi_p : C' \to C'$ acts non-trivially. Finally, the map $\phi : C' \to C$ defined as $\phi : (x', y') \mapsto (u^2x', u^5y')$ moves the result of π_p back to $C(\mathbb{F}_{p^2})$. Composing these maps into $\psi = \phi\pi\phi^{-1}$ gives $\psi : C \to C$, defined as $\psi : (x, y) \mapsto \left(x^p \cdot (u^{-2})^{p-1}, y^p \cdot (u^{-5})^{p-1}\right)$; notice that $10 \mid p-1$ and $u^{10} \in \mathbb{F}_{p^2}$ together imply

that this map is defined over $\mathbb{F}_{p^2}$. Since we use $p \equiv 3 \bmod 4$ and construct $\mathbb{F}_{p^2}$ as $\mathbb{F}_{p^2} = \mathbb{F}_p[i]/(i^2+1)$, we have $z^p = \bar{z}$ for all $z \in \mathbb{F}_{p^2}$, where $\bar{z}$ denotes the *complex conjugate* of z. This ψ map on $C/\mathbb{F}_{p^2}$ extends to give an endomorphism on $\mathrm{Jac}(C)$, given (for general divisors) as

$$\psi : (u_1, u_0, v_1, v_0) \mapsto (\alpha \cdot \bar{u_1},\ \beta \cdot \bar{u_0},\ \gamma \cdot \bar{v_1},\ \delta \cdot \bar{v_0}), \tag{1}$$

where $\alpha = u^{-2(p-1)}$, $\beta = u^{-4(p-1)}$, $\gamma = u^{-3(p-1)}$ and $\delta = u^{-5(p-1)}$ are all precomputed constants in $\mathbb{F}_{p^2}$. Besides the conjugations which are almost for free, it follows that the cost of computing ψ on general divisors is 4 $\mathbb{F}_{p^2}$-multiplications, and it is easily verified that the minimal polynomial of ψ on $\mathrm{Jac}(C)$ is $\Phi_{20}(t) = t^8 - t^6 + t^4 - t^2 + 1$ [17, §5].

Remark 1 (Higher powers of ψ). Scalar decompositions of dimension greater than 2 require the computation of higher powers of ψ on divisors. In all of our cases, applying ψ^i with $i > 1$ costs no more than applying ψ itself: we simply have a different tuple of 4 precomputed constants $(\alpha_i, \beta_i, \gamma_i, \delta_i) \in \mathbb{F}_{p^2}^4$ that allow us to compute ψ^i as in Eq. (1). In fact, applying even powers of ψ is always cheaper than odd powers, since for ψ^{2j} we always have $(\alpha_{2j}, \beta_{2j}, \gamma_{2j}, \delta_{2j}) \in \mathbb{F}_p^4$, so the multiplications required in (1) are now by base field elements. Additionally, for ψ^{2j}, we also have $\delta_{2j} = (-1)^j$ which saves one such multiplication, and finally for even powers of ψ the complex conjugations undo themselves, which saves us performing negations. For 8-GLV/GLS, we need to apply powers of ψ up to ψ^7, so we bear in mind the following order of preference (from cheapest to most expensive): (i) ψ^4, (ii) $\{\psi^2, \psi^6\}$, and (iii) $\{\psi, \psi^3, \psi^5, \psi^7\}$.

4.2 Decomposing the Scalar

Let r be a large prime factor that divides the Jacobian group order of a BK curve $C/\mathbb{F}_{p^2}$ and let D be a divisor of order r on $\mathrm{Jac}(C)$. Since the minimal polynomial of ψ is $\Phi_{20}(t)$ (see Section 4.1), it follows that $\psi(D) = [\lambda]D$ where $\lambda < r \in \mathbb{Z}$ is a root of $t^8 - t^6 + t^4 - t^2 + 1 \equiv 0 \bmod r$. Park, Jeong and Lim [42] gave a simple algorithm that achieves GLV/GLS decompositions through division in the ring $\mathbb{Z}[\psi]$. The first step in this algorithm is to precompute a short vector in the GLV lattice L, which (in our 8-dimensional case) involves finding a short $a = (a_0, \ldots, a_7) \in \mathbb{Z}^8$ in the lattice whose basis (matrix) has leading diagonal $(r, 1, \ldots, 1) \in \mathbb{Z}^8$ and first column $(r, -\lambda, \ldots, -\lambda^7) \in \mathbb{Z}^8$, and where all other entries are zero. We then set $\alpha = \sum_{i=0}^{7} a_i \cdot \psi^i$ and compute a quotient/remainder pair corresponding to the division k/α in $\mathbb{Z}[\psi]$, namely we find the quotient β and the remainder ρ such that $k = \beta\alpha + \rho$ in $\mathbb{Z}[\psi]$. The first observation here is that since $a \in L$, we have $\alpha D = \mathcal{O}$ for all D of order r, and thus $[k]D = \beta\alpha D + \rho D = \rho D$. Since ρ is the remainder in the division by α, its coefficients in $\mathbb{Z}[\psi]$ are also small, so we write $\rho = \sum_{i=0}^{7} k_i \cdot \psi^i$, from which our 8 mini-scalars are $k_0, \ldots, k_7$.

Besides the 8 precomputed "short" constants $a_0, \ldots, a_7$ that must be input into the decomposition routine, there are 9 additional precomputed constants

Algorithm 1. 8-dimensional decomposition of the scalar k on Buhler-Koblitz curves over $\mathbb{F}_{p^2}$ (read the algorithm from left to right and from top to bottom).

Input: The scalar k, the small constants $a_0, \ldots, a_7 \in \mathbb{Z}$ and large constants $b_0, \ldots, b_7, N \in \mathbb{Z}$.

Output: The mini-scalars $k_0, \ldots, k_7$.

$$
\begin{array}{llllllll}
y_0 \leftarrow \lfloor \frac{k \cdot b_0}{N} \rceil, & y_1 \leftarrow \lfloor \frac{k \cdot b_1}{N} \rceil, & y_2 \leftarrow \lfloor \frac{k \cdot b_2}{N} \rceil, & y_3 \leftarrow \lfloor \frac{k \cdot b_3}{N} \rceil, & y_4 \leftarrow \lfloor \frac{k \cdot b_4}{N} \rceil, & y_5 \leftarrow \lfloor \frac{k \cdot b_5}{N} \rceil, & y_6 \leftarrow \lfloor \frac{k \cdot b_6}{N} \rceil, & y_7 \leftarrow \lfloor \frac{k \cdot b_7}{N} \rceil, \\
k_0 \leftarrow k, & u \leftarrow a_0 \cdot y_0, & k_0 \leftarrow k - u, & u \leftarrow a_0 \cdot y_1, & v \leftarrow a_1 \cdot y_0, & u \leftarrow u + v, & k_1 \leftarrow -u, & u \leftarrow a_2 \cdot y_0, \\
v \leftarrow a_0 \cdot y_2, & u \leftarrow u + v, & v \leftarrow a_1 \cdot y_1, & u \leftarrow u + v, & k_2 \leftarrow -u, & u \leftarrow a_3 \cdot y_0, & v \leftarrow a_0 \cdot y_3, & u \leftarrow u + v, \\
v \leftarrow a_1 \cdot y_2, & u \leftarrow u + v, & v \leftarrow a_2 \cdot y_1, & u \leftarrow u + v, & k_3 \leftarrow -u, & u \leftarrow a_0 \cdot y_4, & v \leftarrow a_4 \cdot y_0, & u \leftarrow u + v, \\
v \leftarrow a_1 \cdot y_3, & u \leftarrow u + v, & v \leftarrow a_3 \cdot y_1, & u \leftarrow u + v, & v \leftarrow a_2 \cdot y_2, & u \leftarrow u + v, & k_4 \leftarrow -u, & u \leftarrow a_0 \cdot y_5, \\
v \leftarrow a_5 \cdot y_0, & u \leftarrow u + v, & v \leftarrow a_1 \cdot y_4, & u \leftarrow u + v, & v \leftarrow a_4 \cdot y_1, & u \leftarrow u + v, & v \leftarrow a_2 \cdot y_3, & u \leftarrow u + v, \\
v \leftarrow a_3 \cdot y_2, & u \leftarrow u + v, & k_5 \leftarrow -u, & u \leftarrow a_0 \cdot y_6, & v \leftarrow a_6 \cdot y_0, & u \leftarrow u + v, & v \leftarrow a_1 \cdot y_5, & u \leftarrow u + v, \\
v \leftarrow a_5 \cdot y_1, & u \leftarrow u + v, & v \leftarrow a_2 \cdot y_4, & u \leftarrow u + v, & v \leftarrow a_4 \cdot y_2, & u \leftarrow u + v, & v \leftarrow a_3 \cdot y_3, & u \leftarrow u + v, \\
k_6 \leftarrow -u, & u \leftarrow a_0 \cdot y_7, & v \leftarrow a_7 \cdot y_0, & u \leftarrow u + v, & v \leftarrow a_1 \cdot y_6, & u \leftarrow u + v, & v \leftarrow a_6 \cdot y_1, & u \leftarrow u + v, \\
v \leftarrow a_2 \cdot y_5, & u \leftarrow u + v, & v \leftarrow a_5 \cdot y_2, & u \leftarrow u + v, & v \leftarrow a_3 \cdot y_4, & u \leftarrow u + v, & v \leftarrow a_4 \cdot y_3, & u \leftarrow u + v, \\
k_7 \leftarrow -u, & u \leftarrow a_1 \cdot y_7, & v \leftarrow a_7 \cdot y_1, & u \leftarrow u + v, & v \leftarrow a_2 \cdot y_6, & u \leftarrow u + v, & v \leftarrow a_6 \cdot y_2, & u \leftarrow u + v, \\
v \leftarrow a_3 \cdot y_5, & u \leftarrow u + v, & v \leftarrow a_4 \cdot y_4, & u \leftarrow u + v, & v \leftarrow a_5 \cdot y_3, & u \leftarrow u + v, & k_0 \leftarrow k_0 + u, & k_2 \leftarrow k_2 - u, \\
k_4 \leftarrow k_4 + u, & k_6 \leftarrow k_6 - u, & u \leftarrow a_6 \cdot y_3, & v \leftarrow a_7 \cdot y_2, & u \leftarrow u + v, & v \leftarrow a_3 \cdot y_6, & u \leftarrow u + v, & v \leftarrow a_4 \cdot y_5, \\
u \leftarrow u + v, & v \leftarrow a_5 \cdot y_4, & u \leftarrow u + v, & v \leftarrow a_2 \cdot y_7, & u \leftarrow u + v, & k_1 \leftarrow k_1 + u, & k_3 \leftarrow k_3 - u, & k_5 \leftarrow k_5 + u, \\
k_7 \leftarrow k_7 - u, & u \leftarrow a_4 \cdot y_6, & v \leftarrow a_5 \cdot y_5, & u \leftarrow u + v, & v \leftarrow a_6 \cdot y_4, & u \leftarrow u + v, & v \leftarrow a_7 \cdot y_3, & u \leftarrow u + v, \\
v \leftarrow a_3 \cdot y_7, & u \leftarrow u + v, & k_0 \leftarrow k_0 + u, & u \leftarrow a_7 \cdot y_4, & v \leftarrow a_6 \cdot y_5, & u \leftarrow u + v, & v \leftarrow a_4 \cdot y_7, & u \leftarrow u + v, \\
v \leftarrow a_5 \cdot y_6, & u \leftarrow u + v, & k_1 \leftarrow k_1 + u, & u \leftarrow a_7 \cdot y_5, & v \leftarrow a_6 \cdot y_6, & u \leftarrow u + v, & v \leftarrow a_5 \cdot y_7, & u \leftarrow u + v, \\
k_2 \leftarrow k_2 + u, & u \leftarrow a_7 \cdot y_6, & v \leftarrow a_6 \cdot y_7, & u \leftarrow u + v, & k_3 \leftarrow k_3 + u, & u \leftarrow a_7 \cdot y_7, & k_4 \leftarrow k_4 + u. &
\end{array}
$$

that aid a faster division [42]. Let $g(t) \in \mathbb{Z}[t]$ be the minimal polynomial of $\alpha \in \mathbb{Z}[\psi]$ with constant term N, so that we can write it as $g(t) = t \cdot h(t) + N$. We precompute $\hat{\alpha} = -h(\alpha) = \sum_{i=0}^{7} b_i \psi^i$, which is N/α in $\mathbb{Z}[\psi]$. Along with the scalar k, we input the 8 values $a_0, \ldots, a_7$, the 8 values $b_0, \ldots, b_7$, and N into the decomposition algorithm from [42, §5.2], which we present in three-operand form in Algorithm 1. The first line of Algorithm 1 shows the most non-trivial part of decomposing k on the fly, while the rest of the algorithm is straightforward. For $i = 0, \ldots, 7$, we compute the rounded division $y_i = \lfloor \frac{k \cdot b_i}{N} \rceil$ using only integer operations. We find the smallest b' such that $N < 2^{bb'}$, where b is the width of the machine word-size (32 or 64 in practice). We then precompute $\ell_i = \lfloor \frac{2^{bb'} \cdot b_i}{N} \rceil \geq 0$, so that the division can now be computed as $y_i = \lfloor \frac{\ell_i \cdot k}{2^{bb'}} \rfloor$. The division by $2^{bb'}$ comes for free: it can be implemented by a shift of the machine words of the results. Depending on the sign of k, the result can be off by one due to the rounding, but in practice this does not influence the size of the mini-scalars.

4.3 Constructing the Lookup Table

After the scalar k is decomposed into 8 mini-scalars $k_i < 2^m$, each corresponding to the divisor $D_i = \mathtt{sign}(k_i) \cdot \psi^i(D)$, following the standard approach [17,33,8] (for 2- and 4-dimensional decompositions) would mean computing the scalar multiplication by first precomputing a lookup table $L[i] = \sum_{\ell=0}^{7} \left(\lfloor \frac{i}{2^\ell} \rfloor \bmod 2 \right) \cdot D_\ell$, for $0 \leq i < 2^8$. When simultaneously processing the j^{th} bit of each of the mini-scalars, the precomputed multiple $L[i]$ is added to the accumulator of the main loop, for $i = \sum_{\ell=0}^{7} 2^\ell \left(\lfloor \frac{k_\ell}{2^j} \rfloor \bmod 2 \right)$. The advantage here is that only one doubling and one addition are used for each of the m bits in the mini-scalar. The precomputation phase, computing the entries of the $L[i]$, is relatively inexpensive for 2- and 4-dimensional GLV/GLS. In the setting of 8-GLV/GLS however, computing these $2^8 = 256$ entries is computationally significant: roughly speaking,

Table 2. Generating the lookup table in constant time for 8-dimensional GLV/GLS, where the divisors D_i are computed efficiently from D and k_i sequentially as follows (the cost is stated in the table): $D_0 = D$, $D_1 = \phi(D_0)$, $D_i = \phi^2(D_{i-2})$ for $i \in \{2, 3\}$, $D_i = \phi^4(D_{i-4})$ for $i \in \{4, 5, 6, 7\}$, $D_i = \mathtt{sign}(k_i) \cdot D_i$ for $0 \leq i < 8$. The second argument in the mixed sums is the affine divisor.

operation	D_3	D_2	D_1	D_0	op.
$T_1[0] \leftarrow \mathcal{O}$	0	0	0	0	-
$T_1[1] \leftarrow D_0$	0	0	0	1	-
$T_1[2] \leftarrow D_1$	0	0	1	0	ψ
$T_1[3] \leftarrow T_1[1] + T_1[2]$	0	0	1	1	AFF
$T_1[4] \leftarrow D_2$	0	1	0	0	ψ^2
$T_1[5] \leftarrow T_1[1] + T_1[4]$	0	1	0	1	AFF
$T_1[6] \leftarrow T_1[2] + T_1[4]$	0	1	1	0	AFF
$T_1[7] \leftarrow T_1[6] + T_1[1]$	0	1	1	1	MIX
$T_1[8] \leftarrow D_3$	1	0	0	0	ψ^2
$T_1[9] \leftarrow T_1[1] + T_1[8]$	1	0	0	1	AFF
$T_1[10] \leftarrow T_1[2] + T_1[8]$	1	0	1	0	AFF
$T_1[11] \leftarrow T_1[10] + T_1[1]$	1	0	1	1	MIX
$T_1[12] \leftarrow T_1[8] + T_1[4]$	1	1	0	0	AFF
$T_1[13] \leftarrow T_1[12] + T_1[1]$	1	1	0	1	MIX
$T_1[14] \leftarrow T_1[12] + T_1[2]$	1	1	1	0	MIX
$T_1[15] \leftarrow T_1[14] + T_1[1]$	1	1	1	1	MIX

operation	D_7	D_6	D_5	D_4	op.
$T_2[0] \leftarrow \mathcal{O}$	0	0	0	0	-
$T_2[1] \leftarrow D_4$	0	0	0	1	ψ^4
$T_2[2] \leftarrow D_5$	0	0	1	0	ψ^4
$T_2[3] \leftarrow T_2[1] + T_2[2]$	0	0	1	1	AFF
$T_2[4] \leftarrow D_6$	0	1	0	0	ψ^4
$T_2[5] \leftarrow T_2[1] + T_2[4]$	0	1	0	1	AFF
$T_2[6] \leftarrow T_2[2] + T_2[4]$	0	1	1	0	AFF
$T_2[7] \leftarrow T_2[6] + T_2[1]$	0	1	1	1	MIX
$T_2[8] \leftarrow D_7$	1	0	0	0	ψ^4
$T_2[9] \leftarrow T_2[1] + T_2[8]$	1	0	0	1	AFF
$T_2[10] \leftarrow T_2[2] + T_2[8]$	1	0	1	0	AFF
$T_2[11] \leftarrow T_2[10] + T_2[1]$	1	0	1	1	MIX
$T_2[12] \leftarrow T_2[8] + T_2[4]$	1	1	0	0	AFF
$T_2[13] \leftarrow T_2[12] + T_2[1]$	1	1	0	1	MIX
$T_2[14] \leftarrow T_2[12] + T_2[2]$	1	1	1	0	MIX
$T_2[15] \leftarrow T_2[14] + T_2[1]$	1	1	1	1	MIX

constructing this full-sized lookup table would be as expensive as computing the scalar multiplication in the generic way (i.e. not using endomorphisms). An observation is that in practice m is usually small ($m < 34$), so that we do not need to precompute the entire table and we can compute the required entries on-the-fly. Unfortunately, computing a random table element might require multiple additions (in the worst case) and no performance gain can be expected when using this approach.

We present two different approaches that solve this problem (which can both be seen as an extension to the approach described in [32], but in the special case of two tables). Both approaches generate *two* lookup tables consisting of $2^4 = 16$ elements each. So instead of computing the single large table L, one can compute two significantly smaller tables T_1 and T_2 such that $T_1[i] = \sum_{\ell=0}^{3}(\lfloor \frac{i}{2^\ell} \rfloor \bmod 2) \cdot D_\ell$ and $T_2[i] = \sum_{\ell=0}^{3}(\lfloor \frac{i}{2^\ell} \rfloor \bmod 2) \cdot D_{\ell+4}$, for $0 \leq i < 2^4$. This has the advantage of significantly lowering the precomputation cost of the tables, but increases the number of "per bit" curve additions from one to two when processing the mini-scalars. The two methods we present differ in how the tables are generated: the first approach is slightly slower than the second, but has the advantage that it runs in constant time.

The Constant-Time Approach. The straight-forward approach to generate the two lookup tables T_1 and T_2 is to first compute $T_i[j]$ for $i \in \{1, 2\}$ and $j \in \{1, 2, 4, 8\}$ using (at most) the ψ map for each computation – we prioritize higher even powers of ψ following Remark 1. Next, the other elements are computed

Table 3. Summary of costs for a single 8-GLV/GLS scalar multiplication, where $\max\{|k_i|\} < 2^m$. The left side of the table gives the cost and number of occurrences of the 5 divisor operations used for the table generation (T) and when computing the scalar (S), which are combined to give a total cost of 8-GLV/GLS in terms of m. For each of the implementations in this work, the right side of the table uses the average value of m to give the average number of multiplications, squarings and additions required in 8-GLV/GLS. While the costs reported correspond to the simple, constant-time precomputation strategy, the final column on the right side gives the number of additions (both mixed and affine) that are replaced with ψ's if the faster precomputation strategy is employed. All averages were taken over 10 million scalar decompositions.

div	formulas found in	cost per. operation	T	S	curve	av. m	average cost [**m**, **s**, **a**]	av. ψ's
`DBL`	[8, Alg. 1]	$36\mathbf{m} + 6\mathbf{s} + 34\mathbf{a}$	-	m	p_{611}	26.43	$[4039, 490, 3070]$	9.15
`ADD`	[8, Alg. 2]	$44\mathbf{m} + 4\mathbf{s} + 29\mathbf{a}$	-	$2m$	Mont. (i)	27.53	$[4176, 505, 3171]$	8.90
`MIX`	[8, Alg. 3]	$37\mathbf{m} + 5\mathbf{s} + 29\mathbf{a}$	12	-	Mont. (ii)	31.10	$[4618, 555, 3499]$	8.41
`AFF`	[9]	$29\mathbf{m} + 6\mathbf{s} + 29\mathbf{a}$	10	-	NIST (i)	27.84	$[4214, 509, 3199]$	9.25
ψ	Eq. (1)	$4\mathbf{m}$	7	-	NIST (ii)	31.71	$[4693, 563, 3554]$	8.41
total: $(124m + 762)\mathbf{m} + (14m + 120)\mathbf{s} + (92m + 638)\mathbf{a}$								

as $T_i[j] = T_i[k] + T_i[j-k]$ for $j > 1$ and $k < j$, and where k is chosen so that the fastest possible formulas can be applied each time. Namely, elements that are obtained by using an addition become projective divisors, whilst elements that are in $T_{1,2}[j]$ for $j \in \{1, 2, 4, 8\}$ (which are computed using the ψ map) are affine. Adding two affine divisors together to give a projective divisor is faster than performing a mixed addition between an affine and projective divisor, so we prioritize this affine-only addition where possible. We modified the formulas for the mixed-addition operation to formulas for an affine-affine addition operation, which are given in the full version of this paper [9]. Compared to mixed-additions, this lowers the required number of multiplications in $\mathbb{F}_{p^2}$ from 37 to 29. We denote the operations of projective doubling, projective addition, mixed addition and addition between two affine divisors by `DBL`, `ADD`, `MIX` and `AFF` respectively. Table 2 outlines our approach to compute both lookup tables in constant time. Table 3 summarizes the total cost for both the precomputation of the lookup tables and the computation of the scalar of the 8-GLV/GLS routine as a function of the maximum bit-length m of the mini-scalars k_i. We use **m**, **s** and **a** to denote the costs of computing multiplications, squarings and additions in $\mathbb{F}_{p^2}$ respectively.

Using ψ to Speed Up Precomputations. If we are not concerned with implementations which need to run in constant time and aim to optimize for performance only, then the endomorphism ψ can be used to accelerate the computation of T_1 and T_2. The reason we can not use ψ in the same way for each scalar is that its usefulness and applicability depends on the signs of the k_i, which change each time. We use an example to illustrate: define $s_i = \texttt{sign}(k_i) \in \{-1, +1\}$, and suppose that after computing $D_0, \ldots, D_7$ (which are negated according

to the signs of $k_0, \ldots, k_7$), we compute $T_1[3] \leftarrow T_1[1] + T_1[2] = D_0 + D_1$. When computing $T_1[6]$, which is usually computed using an affine addition as $T_1[6] = T_1[2] + T_1[4] = D_1 + D_2$, we can possibly use ψ to compute $D_1 + D_2$. If the signs s_0, s_1, s_2 are equal then $T_1[6] = D_1 + D_2 = \psi(D_0 + D_1)$, while if s_0 and s_2 are equal and $s_1 = -s_0$, then $T_1[6] = D_1 + D_2 = -\psi(D_0 + D_1)$. Alternatively, if $s_0 \neq s_2$, then we still need (at least) one addition on top of $\psi(D_0 + D_1)$ to compute $D_1 + D_2$ and so using the original addition between $T_1[2]$ and $T_1[4]$ is preferred.

In the full version [9] we outline the complete strategy which exhausts each possibility of using ψ to recycle prior computations before resorting to a divisor addition. As in the above example, the usefulness of previous values is completely dependent on the combinations of the associated signs. As we proceed further into the algorithm, the chances of reusing previous computations generally increases. For example, $T_2[12]$ would ordinarily require the addition $T_2[12] = T_2[8] + T_2[4] = D_7 + D_6$, but it could also possibly be computed as any of $\psi(D_6 + D_5)$, $\psi^2(D_5 + D_4)$, $\psi^4(D_3 + D_2)$, $\psi^5(D_2 + D_1)$ or $\psi^6(D_1 + D_0)$, depending on whether the associated s_i align favorably. Again, we prioritize the possible application of even powers of ψ according to the hierarchy discussed in Remark 1. We note that anytime ψ is used to recycle previously computed sums, they are now acting on projective (instead of affine) divisors. This requires an updated description of ψ, which is given as $\psi : (U_1 : U_0 : V_1 : V_0 : Z) \mapsto \left(\alpha \cdot \bar{U_1} : \beta \cdot \bar{U_0} : \gamma \cdot \bar{V_1} : \delta \cdot \bar{V_0} : \bar{Z}\right)$, for which the only difference from the affine version in Eq. (1) is that the Z coordinate must also be conjugated. We point out that Remark 1 applies identically to the projective case. Of the 22 additions that would otherwise be required, the final column in the right part of Table 3 gives the average number of additions that are replaced by ψ's in the six different 8-GLV/GLS scenarios we implemented. In all cases this gives over a 30% speedup when constructing the lookup table.

5 Arithmetic

In this paper we are concerned with arithmetic modulo quadratic extensions of primes $p < 2^{64}$ to realize scalar multiplications in $\mathrm{Jac}_C(\mathbb{F}_{p^2})$. We optimize this arithmetic on two different levels: on the one hand the extension field arithmetic in $\mathbb{F}_{p^2}$ is optimized in terms of multiplications in $\mathbb{F}_p$, and on the other hand we aim to optimize the multiplications in $\mathbb{F}_p$ by choosing p such that modular reduction is particularly efficient. On architectures where the 64-bit modulus p fits in a single machine word, the modular multiplication can be computed by doing the multiplication first, followed by a NIST-like reduction [44,46]. Other popular embedded platforms, like the ARM, have a smaller machine word size of 32 bits. Since representing the prime p requires two such words, other techniques (besides the NIST-like reduction) might be attractive to explore. Following the observations from [8], we choose the primes p to be Montgomery-friendly to accelerate the implementation of the modular arithmetic on such 32-bit platforms. Since the use of Montgomery-friendly primes only makes sense when the prime can be

Algorithm 2. This algorithm, including Line 1 and Line 3, computes the radix-2^b interleaved Montgomery multiplication [36] ($\text{MontMul}(A, B, p) = A \cdot B \cdot 2^{-bn} \bmod p$) for an n-word modulus p. Excluding Line 1 and Line 3 gives the algorithm for computing the radix-2^b Montgomery reduction only ($\text{MontRed}(C, p) = C \cdot 2^{-bn} \bmod p$).

Input: $\begin{cases} (A = \sum_{i=0}^{n-1} a_i 2^{bi}, B) \text{ or } C \text{ and } p, \mu \text{ such that } 0 \leq a_i < 2^b, 0 \leq A \leq S_0 < 2^{bn}, \\ 0 \leq B \leq S_1 < 2^{bn}, 0 \leq C \leq S_1 S_2 < 2^{2bn},\ 2^{b(n-1)} \leq p < 2^{bn},\ 2 \nmid p, \\ \mu = -p^{-1} \bmod 2^b, \end{cases}$

Output: $\begin{cases} (C' \equiv A \cdot B \cdot 2^{-bn} \bmod p) \text{ or } (C' \equiv C \cdot 2^{-bn} \bmod p) \\ \text{such that } 0 \leq C' < r_{(b,n)}(S_0 S_1, p) \end{cases}$

1: $[C \leftarrow 0]$
2: **for** $i = 0$ to $n - 1$ **do**
3: $\quad [C \leftarrow C + a_i \cdot B]$
4: $\quad q \leftarrow \mu \cdot C \bmod 2^b,\ C \leftarrow (C + q \cdot p)/2^b$
5: **return** $C' \leftarrow C$

represented by two or more machine words, our approach for 64-bit architectures follows the more conventional NIST-like reduction.

5.1 Modular Arithmetic

NIST-Like Reduction. It is well-known that modular reduction can be computed efficiently, without using any multiplications, when the modulus has a special form. Typically, the modular multiplication and the modular reduction are computed sequentially. An example of a family of such primes are generalized Mersenne primes, whose adoption usually results in significant performance gains; this is why NIST has standardized multiple instances of such primes [46]. Let us illustrate the basic idea with $p_{611} = 2^{61} - 1$, which belongs to this class of primes. Computing the modular multiplication $c \equiv a \cdot b \bmod p_{611}$ with $0 \leq a, b < p_{611}$ can be done by first computing the multiplication and shifting this value (note that the result still fits in a 128-bit data-type) as $t = (2^3 \cdot a) \cdot b$. Due to the special form of p_{611}, we have $t = t_1 \cdot 2^{64} + t_0 \equiv t_1 \cdot 2^{64} + t_0 - t_1 \cdot 2^3 \cdot p_{611} \equiv t_0 + 2^3 \cdot t_1 \bmod p_{611}$, for $0 \leq t_0, t_1 < 2^{64}$, and hence we can compute the reduction as $\left(\lfloor t/2^{64} \rfloor + (t \bmod 2^{64})/2^3 \right) \bmod p_{611}$. Since $0 \leq \lfloor t/2^{64} \rfloor, (t \bmod 2^{64})/2^3 < 2^{61}$, we can use these integers as input to a modular addition to reduce the result properly to the range $[0, p_{611}\rangle$. For numbers of the form $2^x - 1$, modular addition is especially efficient, since if $c = a + b$, where $0 \leq a, b < 2^x - 1$, then $c' = \lfloor \frac{c}{2^x} \rfloor + c - 2^x \equiv c \pmod{2^x - 1}$, where c' is properly reduced and can be computed using only a `shift`, an `add` and a `bit-reset` instruction (and possibly data movements).

Montgomery Arithmetic. Montgomery proposed a new way of computing modular multiplication in the mid 1980s [36]. The idea behind Montgomery multiplication is to replace the relatively expensive divisions by computationally inexpensive logical shifts on computers, lowering the computational complexity by a constant factor compared to the classical method. We present the algorithm for a computer platform which works on b-bit ($b > 2$) words: i.e. we use

a 2^b-radix system. Montgomery multiplication modulo an n-word odd moduli p, $2^{(n-1)b} \leq p < 2^{nb}$, is computed by transforming each of the operands to its Montgomery residue $\tilde{A} = A \cdot 2^{bn} \bmod p$. Montgomery multiplication is defined as $\tilde{C} \equiv \tilde{A} \cdot \tilde{B} \cdot 2^{-bn} \equiv C \cdot 2^{bn} \bmod p$. Algorithm 5.1, including the lines in brackets, outlines interleaved Montgomery multiplication, while if the bracketed lines are excluded this computes the Montgomery reduction only. Note that modular addition and subtraction can be done in the usual way when working with Montgomery residues since $\tilde{A} \pm \tilde{B} \equiv (A \pm B) \cdot 2^{bn} \equiv \widetilde{A \pm B} \pmod p$. The result of the Montgomery multiplication of two positive integers $\tilde{A} \leq S_0$ and $\tilde{B} \leq S_1$ can be bounded by $r_{(b,n)}(S_0 \cdot S_1, p) = \frac{S_1 S_2}{2^{bn}} + p$. Hence, if both inputs are bounded by 2^{bn}, then the result is at most $r_{(b,n)}(2^{2bn}, p) = 2^{bn} + p$: a conditional subtraction with p is required when the output is required to be less than 2^{bn}. It follows that if both inputs are bounded by 2^{bn-1} and $2^{b(n-1)} \leq p < 2^{bn-2}$, then this conditional subtraction can be omitted since $r_{(b,n)}(2^{2(bn-1)}, p) = 2^{bn-2} + p < 2^{bn-1}$, and the output of Montgomery multiplication can be reused as input directly (this is the idea behind subtraction-less Montgomery multiplication [47]).

Montgomery-Friendly Arithmetic. The idea behind Montgomery-friendly primes [31,28,1,25,8] is to reduce the number of multiplications and registers used by taking $\mu = -p^{-1} \bmod 2^b = \pm 1$; this is achieved when $p \equiv \mp 1 \bmod 2^b$. Note that all NIST primes, as standardized in [46], have this property for $b \leq 32$. The number of multiplications can be reduced further when the $(n-1)$ most significant words of p have a special form, such that multiplication by p can be transformed into a sequence of shifts and additions or subtractions (just as in the NIST-like reduction). For $b = 32$, examples of Montgomery-friendly primes are those primes of the form $(2^{31} - c) \cdot 2^{32} - 1$, with $0 \leq c < 2^{31}$, as mentioned in Section 3. Here we intentionally use 63-bit primes (instead of the full double-word length of 64 bits) to allow accumulation in the Montgomery reduction without using an additional word. Note that the Mersenne prime used in our NIST-like reduction example is Montgomery-friendly as well, since $p_{611} = 2^{61} - 1 = 2^{29} \cdot 2^{32} - 1$.

5.2 Extension Field Arithmetic

Arithmetic in $\mathbb{F}_{p^2} = \mathbb{F}_p[i]/(i^2 + 1)$ is realized using arithmetic operations from $\mathbb{F}_p$. For instance, the result of multiplying two elements $a_0 + a_1 i, b_0 + b_1 i \in \mathbb{F}_{p^2}$ is $(a_0 b_0 - a_1 b_1) + (a_0 b_1 + a_1 b_0)i \in \mathbb{F}_{p^2}$. This can be achieved using four $\mathbb{F}_p$-multiplications, one $\mathbb{F}_p$-subtraction and one $\mathbb{F}_p$-addition or, when using a single level of Karatsuba, using three $\mathbb{F}_p$-multiplications, two $\mathbb{F}_p$-subtractions and three $\mathbb{F}_p$-additions. To optimize this further, we follow the lazy-reduction techniques described in [3], where the idea is to delay the modular reductions until the end of the computation. This has the advantage of reducing the number of reductions at the cost of performing the intermediate additions and subtractions on numbers of twice the bit-length. When using Karatsuba, this approach is outlined in Algorithm 3 (where we abbreviate $r_{(b,n)}$ to r), together with the bounds on all intermediate values (given the bounds S_0 and S_1 on the inputs). In order

Algorithm 3. $\mathbb{F}_{p^2}$ multiplication using Karatsuba and lazy reduction following [3].

Input: $\begin{cases} (a_0 + a_1 i), (b_0 + b_1 i) \in \mathbb{F}_{p^2}, \text{ with} \\ 0 \leq a_0, a_1 < S_0, 0 \leq b_0, b_1 < S_1. \\ \tilde{m} = m \times p \text{ such that } \tilde{m} \geq S_0 S_1. \end{cases}$

Output: $(c_0 + c_1 i) = (a_0 + a_1 i)(b_0 + b_1 i)$

1: $T_0 \leftarrow a_0 \times b_0$ $(< S_0 S_1)$
2: $T_1 \leftarrow a_1 \times b_1$ $(< S_0 S_1)$
3: $t_0 \leftarrow a_0 + a_1$ $(< 2S_0)$
4: $t_1 \leftarrow b_0 + b_1$ $(< 2S_1)$
5: $T_2 \leftarrow t_0 \times t_1$ $(< 4S_0 S_1)$
6: $T_3 \leftarrow T_2 - (T_0 + T_1)$ $(< 2S_0 S_1)$
7: $c_1 \leftarrow \text{MontRed}(T_3)$ $(< r(2S_0 S_1, p))$
8: $T_4 \leftarrow T_0 + \tilde{m} - T_1$ $(< S_0 S_1 + \tilde{m})$
9: $c_0 \leftarrow \text{MontRed}(T_4)$ $(< r(S_0 S_1 + \tilde{m}, p))$

Algorithm 4. $\mathbb{F}_{p^2}$ squaring.

Input: $\begin{cases} (a_0 + a_1 i) \in \mathbb{F}_{p^2}, \\ \text{with } 0 \leq a_0, a_1 < S_0. \end{cases}$

Output: $(c_0 + c_1 i) = (a_0 + a_1 i)^2$

1: $T_0 \leftarrow a_0 + a_1$ $(< 2S_0)$
2: $T_1 \leftarrow a_0 + p - a_1$ $(< 2S_0)$
3: $c_0 \leftarrow \text{MontMul}(T_0, T_1)$ $(< r(4S_0^2, p))$
4: $T_3 \leftarrow 2a_0$ $(< 2S_0)$
5: $c_1 \leftarrow \text{MontMul}(T_3, a_1)$ $(< r(2S_0^2, p))$

to avoid working with negative numbers, we also require an additional precomputed input value $\tilde{m}$, which is a multiple of p such that $\tilde{m} \geq S_0 S_1$. In practice the bounds on the input are chosen such that both $2S_0$ and $2S_1$ are less 2^{bn}, to avoid making the multiplication $t_0 \times t_1$ in Line 5 of Algorithm 3 work on more computer words. We found that the approach outlined Algorithm 3 (using Karatsuba and postponing the reductions) to be preferable on the 32-bit ARM Cortex-A8 platform. However, on our 64-bit Ivy Bridge platform, calculating the $\mathbb{F}_{p^2}$ multiplication is more efficient using the "naive" schoolbook multiplication (but still using the lazy-reduction techniques to postpone the modular reductions). This requires one additional modular multiplication compared to Karatsuba, but lowers the modular additions/subtractions to only two. Due to the relatively low cost ratio between 64-bit modular multiplications and 64-bit additions, it is more efficient to use schoolbook on such 64-bit platforms. Note that due to our representation of $\mathbb{F}_{p^2}$, squaring can be computed using only two $\mathbb{F}_p$ multiplications, since $(a_0 + a_1 i)^2 = (a_0 + a_1)(a_0 - a_1) + 2a_0 a_1 i$. This approach (including the bounds on the output) is given in Algorithm 4.

For computations modulo $p_{611} = 2^{61} - 1$ on the ARM, we choose to use Montgomery multiplication in combination with a conditional final subtraction, since such a subtraction is particularly efficient (see Section 5.1). This has the advantage of allowing us to add (or subtract) numbers *without* reducing them and using them as input, since if $S_0 = 2(2^{61} - 1)$ and $S_1 = (2^{61} - 1)$, then the first Montgomery reduction in Algorithm 3 is bounded by $r_{(4,32)}(8p_{611}^2, p_{611}) - p_{611} < p_{611}$, so that the result is automatically properly reduced. For the second reduction, we could choose $\tilde{m} = (2^{63} + 1) \cdot p_{611}$ such that $r_{(4,32)}(2p_{611}^2, p_{611}) + (2^{63} + 1) \cdot p_{611} - p_{611} < p_{611}$ is also properly reduced. Another possibility is to choose $\tilde{m} = 2^{64} \cdot p_{611}$ to avoid adding the least significant 64 bits of $\tilde{m}$, which reduces the number of required addition instructions. However, in this case we would need one more conditional subtraction, since $r_{(4,32)}(2p_{611}^2, p_{611}) +$

$2^{64} \cdot p_{611} - k \cdot p_{611} < p_{611}$ holds for $k \geq 2$. For the other Montgomery-friendly primes, we performed a similar analysis to minimize the number of reductions after additions and subtractions.

Using Mixed Additions. As outlined in Table 3, mixed divisor additions are significantly faster than using regular (projective) divisor additions. It is a common approach to convert the projective divisors in the lookup table to affine divisors in order to use these faster formulas when computing the scalar multiplication. This can be done efficiently using Montgomery's *simultaneous inversion* method [37]. Supposing there are w such projective divisors in our lookup table(s), the simultaneous inversion method finds the w independent inverses using a single inversion and $3(w-1)$ multiplications. For each of the w projective divisors of the form $(U_1 \colon U_0 \colon V_1 \colon V_0 \colon Z)$, normalization (given Z^{-1}) costs four additional multiplications. Hence, the total cost of converting the entire lookup table to affine coordinates is $(7w-3)\mathbf{m} + \mathbf{I}$, where $\mathbf{I}$ is the cost of an inversion in $\mathbb{F}_{p^2}$. To compute the inverse in $\mathbb{F}_{p^2} = \mathbb{F}_p[i]/(i^2+1)$, we use $(a_0 + a_1 i)^{-1} = a_0/(a_0^2 + a_1^2) + (-a_1/(a_0^2 + a_1^2))i$, which costs, besides the $\mathbb{F}_p$-inversion, two $\mathbb{F}_p$-squarings, two $\mathbb{F}_p$-multiplications, a single $\mathbb{F}_p$-addition and a single $\mathbb{F}_p$-negation. Our implementations on both platforms revealed that it was always preferable to perform this normalization, i.e. that the cost of normalizing the lookup table is outweighed by the savings achieved when processing the scalar.

6 Results and Discussion

We implemented the generic, Kummer and 8-dimensional GLV/GLS (see Section 4) techniques using the different arithmetic approaches (as outlined in Section 5). In this section we use our fastest curves (for comparisons with other work) in two settings: one aims solely for performance (non-constant time) while the other provides some side-channel resistance [30] (i.e. runs in constant time). In Table 4 we summarize all the fastest software scalar multiplication results for genus g curves over both $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ for both 64-bit processors and 32-bit ARM architectures.

High-End 64-Bit Architecture. The 64-bit implementations cover the fastest overall constant time performance numbers [14], the fastest constant time performance numbers for elliptic curves over prime fields by Bernstein [5], the fastest (non-constant time) implementation for elliptic curves by Longa and Sica [33], the fastest constant time (Kummer) and non-constant time (4-GLV) performance numbers on genus 2 curves over prime fields [8] by Bos et al., and the fastest implementation of the NIST curve NIST-p224 by Käsper [27]. Note that all of these curves aim to provide 128-bit security, except the NIST curve which is designed to provide 112-bit security. We ran all of these implementations on the same CPU: an Intel Core i7-3520M (Ivy Bridge) processor at 2893.484 MHz with hyperthreading turned off and over-clocking ("turbo boost") disabled. We

Table 4. Performance comparison of scalar multiplication on an Intel Core i7-3520M Ivy Bridge (IB) and various ARM processors (all our code is run on an Cortex-A8). We state the genus g of the curve, if the implementation runs in constant time (CT) or not, the underlying field K, the security in bits (bit sec) provided by the curves and finally the performance number in 10^3 cycles. The performance numbers for 8-GLV/GLS (which use the non constant-time method for computing the lookup table) are in brackets.

	reference	g	CT	K	bit sec	10^3 cycles
Ivy-Bridge	[14] 4-GLV/GLS	1	✓	$\mathbb{F}_{p^2}$	125	92
	[5] curve25519	1	✓	$\mathbb{F}_p$	126	182
	[8] Kummer	2	✓	$\mathbb{F}_p$	125	117
	[8] 4-GLV	2	×	$\mathbb{F}_p$	125	156
	[27] NISTp-224	1	✓	$\mathbb{F}_p$	112	302
	[33] 2-GLV	1	×	$\mathbb{F}_p$	127	145
	new (special, generic)	2	×	$\mathbb{F}_{p^2}$	103	204
	new (NIST, generic)	2	×	$\mathbb{F}_{p^2}$	110	333
	new (special, Kummer)	2	✓	$\mathbb{F}_{p^2}$	103	108
	new (NIST, Kummer)	2	✓	$\mathbb{F}_{p^2}$	110	167
	new (special, 8-GLV/GLS)	2	×	$\mathbb{F}_{p^2}$	105	100 (92)
	new (NIST, 8-GLV/GLS)	2	×	$\mathbb{F}_{p^2}$	111	146 (136)
ARM	[14] 4-GLV/GLS (Cortex-A9)	1	✓	$\mathbb{F}_{p^2}$	125	417
	[7] curve25519 (Cortex-A8/NEON)	1	✓	$\mathbb{F}_p$	126	527
	[25] twisted Edwards (Cortex-A9)	1	✓	$\mathbb{F}_p$	125	616
	[38] NISTp-224 (Cortex-A8)	1	?	$\mathbb{F}_p$	112	7805
	new (special, generic)	2	×	$\mathbb{F}_{p^2}$	103	1492
	new (Montgomery, generic)	2	×	$\mathbb{F}_{p^2}$	110	1808
	new (special, Kummer)	2	✓	$\mathbb{F}_{p^2}$	103	767
	new (Montgomery, Kummer)	2	✓	$\mathbb{F}_{p^2}$	108	942
	new (special, 8-GLV/GLS)	2	×	$\mathbb{F}_{p^2}$	105	617 (576)
	new (Montgomery, 8-GLV/GLS)	2	×	$\mathbb{F}_{p^2}$	109	859 (810)

either compiled the code on our machine (for [8,27,5]) or used a precompiled binary provided to us by the authors (for [33,14]).

Table 4 includes our fastest constant time implementation (Kummer) and our fastest non-constant time one (8-dimensional GLV/GLS), which will be made publicly available through [6]. A direct comparison to the state-of-the-art performance numbers is difficult; different curves of varying genus defined over different fields are used and most of the curves in Table 4 aim to provide 128-bit security, while our curves aim for the 112-bit security level. Nevertheless, it is clear from our performance numbers that genus 2 curves over quadratic extension fields are competitive (and often faster) in terms of performance, even when taking the security into account. For instance, when compared to the fast implementation of curve NIST-p224 by Käsper [27], also aiming to provide 112-bit security, we are able to reduce the throughput by roughly a factor three. Interestingly, while implementations on the Kummer surface proved to be faster than 4-GLV/GLS implementations on genus 2 curves over 128-bit prime fields [8], our work over quadratic extension fields of 64-bit primes shows that 8-GLV/GLS overtakes $\mathbb{F}_{p^2}$ Kummer implementations in terms of speed.

Low-End 32-Bit Architecture. For our low-end platform we consider the 32-bit ARM platform. More specifically we run our experiments on the BeagleBoard-xM [4], a low-power open-source hardware single-board computer, which contains an DM3730 processor (1 GHz Cortex-A8 ARM core). Unlike the setting of the 64-bit platforms, we were unable to run implementations from Table 4 on our platform since not all implementations were made available; hence, we copied the performance numbers directly from the papers and mention which ARM processor is used. We point out that the fast performance result by Bernstein and Schwabe [7] was obtained using ARM's NEON instruction set (a combined 64- and 128-bit single instruction, multiple data instruction set), a possibility which has not been studied in this nor the other ARM papers mentioned in Table 4. A direct comparison is again difficult in this case because our curves in Table 1 provide a lower level of security. However, compared to the work by Morozov et al. [38] which also targets the 112-bit security level using the standard NIST curves, our numbers are an order of magnitude faster.

7 Conclusions

In this paper we have explored the possibility of using genus 2 curves over quadratic extension fields in cryptography, where the size of ground field fits into a *single* 64-bit word. This setting allows one to use 8-dimensional GLV/GLS scalar decompositions, which we explored in a variety of scenarios. The downside of using primes of this size for genus 2 based cryptography is that there exist faster-than-generic index calculus attacks which affect the security. Nevertheless, we show how to obtain 112-bit security and present performance numbers for both high-end 64-bit architectures and low-end 32-bit ARM platforms.

References

1. Acar, T., Shumow, D.: Modular reduction without pre-computation for special moduli. Technical report, Microsoft Research (2010)
2. Adleman, L., DeMarrais, J., Huang, M.: A subexponential algorithm for discrete logarithms over hyperelliptic curves of large genus over GF(q). Theoretical Computer Science 226(1-2), 7–18 (1999)
3. Aranha, D.F., Karabina, K., Longa, P., Gebotys, C.H., López, J.: Faster explicit formulas for computing pairings over ordinary curves. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 48–68. Springer, Heidelberg (2011)
4. Beagle Board. BeagleBoard-xM System Reference Manual (2013), `http://beagleboard.org/static/BBxMSRM_latest.pdf`
5. Bernstein, D.J.: Curve25519: New Diffie-Hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 207–228. Springer, Heidelberg (2006)
6. Bernstein, D.J., Lange, T. (eds.): eBACS: ECRYPT Benchmarking of Cryptographic Systems, `http://bench.cr.yp.to` (accessed March 1, 2013)
7. Bernstein, D.J., Schwabe, P.: NEON crypto. In: Prouff, E., Schaumont, P. (eds.) CHES 2012. LNCS, vol. 7428, pp. 320–339. Springer, Heidelberg (2012)

8. Bos, J.W., Costello, C., Hisil, H., Lauter, K.: Fast cryptography in genus 2. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 194–210. Springer, Heidelberg (2013)
9. Bos, J.W., Costello, C., Hisil, H., Lauter, K.: High-performance scalar multiplication using 8-dimensional GLV/GLS decomposition. Cryptology ePrint Archive, Report 2013/146 (2013), http://eprint.iacr.org/
10. Buhler, J., Koblitz, N.: Lattice basis reduction, Jacobi sums and hyperelliptic cryptosystems. Bul. of the Australian Mathematical Society 58(1), 147–154 (1998)
11. Diem, C.: The GHS attack in odd characteristic. J. Ramanujan Math. Soc. 18(1), 1–32 (2003)
12. Diem, C.: On the discrete logarithm problem in elliptic curves. Compositio Mathematica 147(01), 75–104 (2011)
13. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Transactions on Information Theory 22(6), 644–654 (1976)
14. Faz-Hernandez, A., Longa, P., Sanchez, A.H.: Keep calm and stay with one (and $p > 3$). Cryptology ePrint Archive, Report 2013/158 (2013)
15. Frey, G.: How to disguise an elliptic curve (Weil descent). Talk at ECC: slides available at http://cacr.uwaterloo.ca/conferences/1998/ecc98/frey.ps (September 1998)
16. Galbraith, S.D.: Weil descent of Jacobians. Discrete Applied Mathematics 128(1), 165–180 (2003)
17. Galbraith, S.D., Lin, X., Scott, M.: Endomorphisms for faster elliptic curve cryptography on a large class of curves. J. Cryptology 24(3), 446–469 (2011)
18. Gallant, R.P., Lambert, R.J., Vanstone, S.A.: Faster point multiplication on elliptic curves with efficient endomorphisms. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 190–200. Springer, Heidelberg (2001)
19. Gaudry, P.: An algorithm for solving the discrete log problem on hyperelliptic curves. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 19–34. Springer, Heidelberg (2000)
20. Gaudry, P.: Fast genus 2 arithmetic based on theta functions. Journal of Mathematical Cryptology JMC 1(3), 243–265 (2007)
21. Gaudry, P.: Index calculus for abelian varieties of small dimension and the elliptic curve discrete logarithm problem. J. Symb. Comput. 44(12), 1690–1702 (2009)
22. Gaudry, P., Hess, F., Smart, N.P.: Constructive and destructive facets of Weil descent on elliptic curves. J. Cryptology 15(1), 19–46 (2002)
23. Gaudry, P., Thomé, E., Thériault, N., Diem, C.: A double large prime variation for small genus hyperelliptic index calculus. Math. Comput. 76(257), 475–492 (2007)
24. Goren, E.Z., Lauter, K.E.: Genus 2 curves with complex multiplication. International Mathematics Research Notices 2012(5), 1068–1142 (2012)
25. Hamburg, M.: Fast and compact elliptic-curve cryptography. Cryptology ePrint Archive, Report 2012/309 (2012), http://eprint.iacr.org/
26. Iijima, T., Momose, F., Chao, J.: Classification of elliptic/hyperelliptic curves with weak coverings against GHS attack without isogeny condition. Cryptology ePrint Archive, Report 2009/613 (2009), http://eprint.iacr.org/
27. Käsper, E.: Fast elliptic curve cryptography in OpenSSL. In: Danezis, G., Dietrich, S., Sako, K. (eds.) FC 2011 Workshops. LNCS, vol. 7126, pp. 27–39. Springer, Heidelberg (2012)
28. Knežević, M., Vercauteren, F., Verbauwhede, I.: Speeding up bipartite modular multiplication. In: Hasan, M.A., Helleseth, T. (eds.) WAIFI 2010. LNCS, vol. 6087, pp. 166–179. Springer, Heidelberg (2010)

29. Koblitz, N.: Elliptic curve cryptosystems. Mathematics of Computation 48(177), 203–209 (1987)
30. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)
31. Lenstra, A.K.: Generating RSA moduli with a predetermined portion. In: Ohta, K., Pei, D. (eds.) ASIACRYPT 1998. LNCS, vol. 1514, pp. 1–10. Springer, Heidelberg (1998)
32. Lim, C.H., Lee, P.J.: More flexible exponentiation with precomputation. In: Desmedt, Y.G. (ed.) CRYPTO 1994. LNCS, vol. 839, pp. 95–107. Springer, Heidelberg (1994)
33. Longa, P., Sica, F.: Four-dimensional Gallant-Lambert-Vanstone scalar multiplication. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 718–739. Springer, Heidelberg (2012)
34. Miller, V.S.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (1986)
35. Momose, F., Chao, J.: Scholten forms and elliptic/hyperelliptic curves with weak Weil restrictions. Cryptology ePrint Archive, Report 2005/277 (2005)
36. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation 44(170), 519–521 (1985)
37. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Mathematics of Computation 48(177), 243–264 (1987)
38. Morozov, S., Tergino, C., Schaumont, P.: System integration of elliptic curve cryptography on an OMAP platform. In: IEEE 9th Symposium on Application Specific Processors – SASP, pp. 52–57. IEEE Computer Society (2011)
39. Nagao, K.-I.: Decomposition attack for the Jacobian of a hyperelliptic curve over an extension field. In: Hanrot, G., Morain, F., Thomé, E. (eds.) ANTS-IX 2010. LNCS, vol. 6197, pp. 285–300. Springer, Heidelberg (2010)
40. National Institute of Standards and Technology. Special publication 800-57: Recommendation for key management part 1: General (revised), `http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf`
41. National Security Agency. The case for elliptic curve cryptography (2009), `http://www.nsa.gov/business/programs/elliptic_curve.shtml`
42. Park, Y.-H., Jeong, S., Lim, J.: Speeding up point multiplication on hyperelliptic curves with efficiently-computable endomorphisms. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 197–208. Springer, Heidelberg (2002)
43. Pollard, J.M.: Monte Carlo methods for index computation (mod p). Mathematics of Computation 32(143), 918–924 (1978)
44. Solinas, J.A.: Generalized Mersenne numbers. Technical Report CORR 99–39, Centre for Applied Cryptographic Research, University of Waterloo (1999)
45. Thériault, N.: Weil descent attack for Kummer extensions. J. Ramanujan Math. Soc. 18(3), 218–312 (2003)
46. U.S. Department of Commerce/National Institute of Standards and Technology. Digital Signature Standard (DSS). FIPS-186-3 (2009), `http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf`
47. Walter, C.D.: Montgomery exponentiation needs no final subtractions. Electronics Letters 35(21), 1831–1832 (1999)