Keep Calm and Stay with One (and $p > 3$)

Armando Faz-Hernández[1], Patrick Longa[2], and Ana H. Sánchez[1]

[1] Computer Science Department, CINVESTAV-IPN, México
{armfaz,asanchez}@computacion.cs.cinvestav.mx
[2] Microsoft Research,
One Microsoft Way, Redmond, WA 98052, USA
plonga@microsoft.com

Abstract. We demonstrate the high-speed computation of core elliptic curve operations with full protection against timing-type side-channel attacks. We use a state-of-the-art GLV-GLS curve in twisted Edwards form defined over a quadratic extension field of large prime characteristic, which supports a four dimensional decomposition of the scalar. We present highly optimized algorithms and formulas for speeding up the different arithmetic layers, including techniques especially suitable for high-speed, side-channel protected computation on GLV-based implementations. Analysis and performance results are reported for modern x64 and ARM processors. For instance, on an Intel Ivy Bridge processor we compute a variable-base scalar multiplication in 94,000 cycles, a fixed-base scalar multiplication in 53,000 cycles using a table of 6KB, and a double scalar multiplication in 118,000 cycles using a table of 3KB. Similarly, on an ARM Cortex-A15 processor we compute a variable-base scalar multiplication in 244,000 cycles, a fixed-base scalar multiplication in 116,000 cycles (table of 6KB), and a double scalar multiplication in 285,000 cycles (table of 3KB). All these numbers and the proposed techniques represent a significant improvement of the state-of-the-art performance of elliptic curve computations. Most remarkably, our optimizations allow us to reduce the cost of adding protection against timing attacks in the computation of variable-base scalar multiplication to around or below 10%.

Keywords. Elliptic curves, scalar multiplication, GLV-GLS curves, twisted Edwards form, side-channel protection, prime fields.

1 Introduction

Given points P and Q on an elliptic curve subgroup of prime order r and integers $k, l \in [1, r-1]$, there are *three* core scalar multiplication variants that are the basis of most elliptic curve-based protocols: kP with P not known in advance (variable-base scenario), kP with P known in advance (fixed-base scenario), and $kP + lQ$ with P not known in advance and Q known in advance (variable/fixed-base double scalar scenario).

Based on work by Gallant-Lambert-Vanstone [11] and Galbraith-Lin-Scott [10], Longa and Sica proposed in ASIACRYPT 2012 [20] a new family of curves, known as GLV-GLS, which are defined over a quadratic extension field and enable a four-dimensional decomposition of the scalar. They also showed how to model the curves in the efficient twisted Edwards form [3] to improve performance even further. The results in [20] set new speed records in the computation of scalar multiplication on x64 processors, specifically in the representative variable-base scenario and for two cases: when protection against timing-type side channel attacks is required and when it is not.

In this work, we push the envelope further and show how to compute core curve operations very efficiently using GLV-GLS curves in twisted Edwards form on x64 and ARM processors for all the fundamental scenarios above and including full protection against timing attacks [15, 6, 2, 24]. Moreover, our scalar multiplication algorithms exhibit regular execution, providing a first layer of protection against certain simple side-channel analysis (SSCA) attacks such as simple power analysis (SPA) [16]. We have not included yet countermeasures against other sophisticated attacks such as differential power analysis (DPA). To showcase the efficiency of our techniques, we specifically target a 251-bit prime order subgroup on an efficient GLV-GLS twisted Edwards curve over $\mathbb{F}_{p^2}$, where p is the pseudo-Mersenne prime $2^{127} - 5997$, which was proposed in [20]. This curve is referred to as `Ted127-glv4`.

Our main contributions can be summarized as follows:

- We revisit the comb method by Feng et al. [7], which was originally intended for the fixed-base scenario, and present an optimized variant intended for GLV-based *variable-base* scalar multiplication that is based on a modified representation that we refer to as GLV-based LSB-set. The new method is shown to be superior to the best methods for computing side-channel protected variable-base scalar multiplication.
- We propose the technique of interleaving ARM and NEON-based field operations to speed up the curve arithmetic on modern ARM processors. We demonstrate the efficacy of new algorithms based on this approach on the underlying quadratic extension field layer of curve `Ted127-glv4`.
- We present efficient algorithms for implementing field and quadratic extension field operations targeting our 127-bit pseudo-Mersenne prime on x64 and ARM platforms. We combine and exploit in novel ways incomplete reduction [27] and lazy reduction [26], expanding techniques by [19]. These optimized operations are then applied to state-of-the-art formulas in the twisted Edwards model [3, 14] to speed up computations in the setting of curves over $\mathbb{F}_{p^2}$.
- We present benchmark results for curve `Ted127-glv4` for all *three* core computations covering fixed-base, variable-base and double scalar multiplication on x64 and ARM processors. Remarkably, we show that the improved algorithms and formulas dramatically reduce the cost of adding protection against timing attacks and set new speed records for protected software for all three core operations.

For instance, a protected variable-based elliptic curve scalar multiplication on curve `Ted127-glv4` runs in 98,000 cycles on an Intel Sandy Bridge machine. This is 28% faster than the state-of-the-art implementation by [20] that computes the same operation in 137,000 cycles. Most notoriously, this is only 7% slower than the state-of-the-art *unprotected* computation by the same authors, which runs in 91,000 cycles. This result not only represents a new speed record for protected software but also marks the first time that a constant-time computation of variable-base scalar multiplication is performed under 100K cycles on an Intel processor. Similar results are obtained for fixed-base and double scalar multiplication, and for ARM processors using the NEON vector engine (see Section 6 for full details).

We remark that, although one of the objectives of this paper is to achieve the highest speed in protected computations on curve `Ted127-glv4`, many optimized techniques and algorithms have broader applicability. For example, algorithms in Section 3 can be used to speed up GLV-based implementations in general, including the recent genus 2 implementations using the GLV method by Bos et al. [5] and the implementation of a GLV-based binary GLS curve by Oliveira et al. [23].

This paper is organized as follows. In Section 2, we describe curve `Ted127-glv4`. In Section 3, we present the optimized algorithms for side-channel protected variable-base, fixed-base and variable/fixed-base double scalar multiplication. We describe our optimized algorithms for field and extension field operations targeting x64 and ARM platforms in Section 4, and our optimized formulas for twisted Edwards point operations over $\mathbb{F}_{p^2}$ in Section 5. Finally, in Section 6, we perform an analysis of the different methods and present benchmark results of the core scalar multiplication scenarios on the targeted x64 and ARM processors.

2 The Curve

For complete details about the GLS method and the 4-dimensional GLV method using GLV-GLS curves, the reader is referred to [9] and [21], respectively.

In our implementations, we use the following GLV-GLS curve in twisted Edwards form [20], referred to as `Ted127-glv4`:

$$E'_{TE}/\mathbb{F}_{p^2} : -x^2 + y^2 = 1 + dx^2y^2, \tag{1}$$

where $\mathbb{F}_{p^2}$ is defined as $\mathbb{F}_p[i]/(i^2 - \beta)$, $\beta = -1$ is a quadratic non-residue in $\mathbb{F}_p$ and $u = 1 + i$ is a quadratic non-residue in $\mathbb{F}_{p^2}$. Also, $p = 2^{127} - 5997$, $d = 170141183460469231731687303715884099728 + 116829086847165810221872975542241037773i$ and $\#E'_{TE}(\mathbb{F}_{p^2}) = 8r$, where r is the 251-bit prime $2^{251} - 2551080634036073366785319215779098244322 95$. E'_{TE} is isomorphic to the Weierstrass curve $E'_W/\mathbb{F}_{p^2} : y^2 = x^3 - 15/2\ u^2x - 7u^3$, which is the quadratic twist of a curve isomorphic to the GLV curve

$E_W/\mathbb{F}_p : y^2 = 4x^3 - 30x - 28$ (see [20, Section 5]). $E'_{TE}/\mathbb{F}_{p^2}$ is equipped with two efficiently computable endomorphisms Φ and Ψ defined over $\mathbb{F}_{p^2}$, which enable a four-dimensional decomposition for any scalar $k \in [1, r-1]$ in the subgroup generated by a point P of order r and, consequently, enable a four-dimensional scalar multiplication given by

$$kP = k_1P + k_2\Phi(P) + k_3\Psi(P) + k_4\Psi\Phi(P), \text{ with } \max_i(|k_i|) < C\, r^{1/4}$$

for some explicit $C > 0$ [20].

Let $\zeta_8 = u/\sqrt{-2a'}$, where $a' = 27u^3(\frac{\sqrt{2}}{2} - 1)$, be a primitive 8th root of unity. The affine formulas for $\Phi(x, y)$ and $\Psi(x, y)$ are given by

$$\Phi(x,y) = \left(-\frac{(\zeta_8^3 + 2\zeta_8^2 + \zeta_8)xy^2 + (\zeta_8^3 - 2\zeta_8^2 + \zeta_8)x}{2y}, \frac{(\zeta_8^2 - 1)y^2 + 2\zeta_8^3 - \zeta_8^2 + 1}{(2\zeta_8^3 + \zeta_8^2 - 1)y^2 - \zeta_8^2 + 1}\right)$$

and

$$\Psi(x,y) = \left(\zeta_8 x^p, \frac{1}{y^p}\right),$$

respectively. It can be verified that $\Phi^2 + 2 = 0$ and $\Psi^2 + 1 = 0$. The formulas in homogeneous projective coordinates can be found in Appendix A.

Note that curve `Ted127-glv4` has $a = -1$ (in the twisted Edwards equation; [3]), corresponding to the most efficient set of formulas proposed by Hisil et al. [14]. Although GLV-GLS curves with suitably chosen parameters when transformed to twisted Edwards form offer roughly the same performance, as discussed in [20], there are certain differences in the cost of formulas for computing the endomorphisms Φ and Ψ. Curve `Ted127-glv4` exhibits relatively efficient formulas for computing the endomorphisms in comparison with other GLV-GLS curves presented in [20]. On the other hand, our selection of the pseudo-Mersenne prime $p = 2^{127} - 5997$ enables efficient field arithmetic by exploiting lazy and incomplete reduction techniques (see Section 4 for full details). Also, since $p \equiv 3 \pmod 4$, -1 is a quadratic non-residue in $\mathbb{F}_p$, which minimizes the cost of multiplication over $\mathbb{F}_{p^2}$ by transforming multiplications by β to inexpensive subtractions.

3 Scalar Arithmetic

In this section, we introduce an efficient algorithm for computing side-channel protected variable-base scalar multiplication in the GLV setting. We also detail the chosen algorithms and our optimizations for the fixed-base and double scalar scenario. The methods for variable and fixed bases are based on the clever signed representations by Feng et al. [7] and Hedabou et al. [13], in combination with Lim-Lee's comb method. The reader is referred to [17] for complete details on the original comb technique.

3.1 Variable-Base Scalar Multiplication

Let t be the bitlength of a given scalar k. Assume that k is partitioned in w consecutive parts of $d = \lceil t/w \rceil$ bits each, padding k with $(dw - t)$ zeros to the left. Let the updated binary representation of k be $(k_{l-1}, k_{l-2}, \ldots, k_0)$, where $l = dw$. Feng et al. [7] proposed a signed representation based on the equivalence $1 \equiv 1\bar{1}\ldots\bar{1}$. Feng et al.'s representation, called least significant bit-set (LSB-set), consists of first applying such transformation to the least significant part of d bits in the scalar and, then, every remaining bit k_i is converted in such a way that output digits b_i for $d \leq i \leq (l-1)$ be in the digit set $\{0, b_{i \bmod d}\}$. This representation is then used in a comb-style execution for computing kP, when P is known in advance. The method offers a first layer of protection against certain SSCA attacks such as SPA since every "digit-column" is nonzero.

In the following, we adapt the fixed-base LSB-set comb method to the computation of side-channel protected *variable-base* scalar multiplication in the GLV setting. First, we propose a variant of the LSB-set representation for this scenario that we call GLV-based LSB-set. In this representation, the first sub-scalar

is recoded to nonzero digits b_i^0 using the equivalence $1 \equiv 1\bar{1}\ldots\bar{1}$. Remaining sub-scalars are then recoded in such a way that output digits at position i are in the set $\{0, b_i^0\}$, i.e., nonzero digits at the same relative position share the sign. An algorithm to obtain this representation is shown as Algorithm 1. Note also that, in contrast to [7, Alg. 4] and [8, Alg. 2], our recoding is simpler and exhibits a regular and constant time execution, making the algorithm resilient to timing attacks.

Algorithm 1 Protected Recoding with Fixed Length for the GLV-based LSB-Set Representation.

Input: m l-bit integers $k_j = (k_{l-1}^j, \ldots, k_0^j)_2$ for $0 \leq j < m$, where k_0 is odd and $l = \lceil r/m \rceil + 1$.
Output: $k_j = (b_{l-1}^j, \ldots, b_0^j)_{\text{LSB-set}}$ for $0 \leq j < m$, where $b_i^0 \in \{1, -1\}$, and $b_i^j \in \{0, b_i^0\}$ for $1 \leq j < m$.

1: $b_{l-1}^0 = 1$
2: **for** $i = 0$ **to** $(l-2)$ **do**
3: $\quad b_i^0 = 2k_{i+1}^0 - 1$
4: **for** $j = 0$ **to** $(m-1)$ **do**
5: $\quad$ **for** $i = 0$ **to** $(l-1)$ **do**
6: $\quad\quad b_i^j = b_i^0 \cdot k_0^j$
7: $\quad\quad k_j = \lfloor k_j/2 \rfloor - \lfloor b_i^j/2 \rfloor$
8: **return** $(b_{l-1}^j, \ldots, b_0^j)_{\text{LSB-set}}$ for $0 \leq j < m$.

The analysis of the correctness of Algorithm 1 follows. First, note that for our recoding to work in the GLV setting with dimension m, each sub-scalar needs to be padded with zeros to the left until reaching the bitlength l, where $l = \lceil r/m \rceil + 1$ and r is the order of the elliptic curve subgroup. Since the sub-scalar k_0 is odd, converting all bit sequences $00\ldots01$ to $1\bar{1}\ldots\bar{1}\bar{1}$ produces a signed nonzero representation with digits in the set $\{1, -1\}$, without changing the original value. Given a sequence $00\ldots01$ of b bits, the recoding is equivalent to set the $(b-1)$-th bit of the output to 1 and then set to $\bar{1}$ the i-th position of the output per each 0 in the $(i+1)$-th position of the input, for $0 \leq i < b-1$. Observe that this transformation is equivalent to set $b_{l-1}^0 = 1$ and compute $b_i^0 = 2k_{i+1}^0 - 1$ for $0 \leq i < l-1$ (Steps 1-3). Now, we analyze k_1 (the explanation easily extends to other sub-scalars). In this case, we want $b_i^1 \in \{0, b_i^0\}$. One can proceed by scanning bit by bit from right to left, dividing by two every time. There are three possible cases to analyze. If $k_0^1 = 0$ in any given iteration i, the original value remains unchanged by outputting $b_i^1 = b_i^0 \cdot k_0^1 = 0$. If $k_0^1 = 1$ and $b_i^0 = 1$ in any given iteration i, the original value remains unchanged by outputting $b_i^1 = b_i^0 \cdot k_0^1 = 1$. Finally, if $k_0^1 = 1$ and $b_i^0 = -1$ in any given iteration i, the original value remains unchanged by replacing 1 by $1\bar{1}$. This is accomplished by outputting $b_i^1 = b_i^0 \cdot k_0^1 = -1$ and adding 1 to the remaining value of the sub-scalar, i.e., performing $k_1 = \lfloor k_1/2 \rfloor + 1$ at Step 7. Note that only in this last case there is a carry that increases the intermediate value of k_1, converting to 0 subsequent bits 1 until the first 0 is hit. By definition the MSB of k_1 is 0, so if there is a carry bit in the $(l-2)$-th iteration (i.e., k_0^1 becomes 1 in the last iteration) the output digit at iteration $(l-1)$ would be $b_{l-1}^1 = b_{l-1}^0 \cdot k_0^1 = 1 \cdot 1 = 1$, otherwise, it would be 0.

We highlight that Algorithm 1 can be implemented very efficiently by exploiting the fact that the only purpose of the recoded digits from the first sub-scalar is to determine the sign of their corresponding digit-columns (see details of Alg. 2 below). Since $k_{i+1}^0 = 0$ and $k_{i+1}^0 = 1$ indicate that the corresponding output digit-column i will be negative and positive, respectively, Step 3 of Algorithm 1 can be reduced to $b_i^0 = k_{i+1}^0$ by assuming the convention $b_i^0 = 0$ to indicate negative and $b_i^0 = 1$ to indicate positive, for $0 \leq i < l$. Following this convention, further efficient simplifications are possible for Steps 6 and 7.

We now present Algorithm 2 for computing variable-base scalar multiplication with the GLV method. The basic idea is to arrange the sub-scalars in matrix form, with sub-scalar k_0 in the least significant row, and then run a simultaneous multi-scalar execution scanning digit-columns from left to right. By using the GLV-based LSB-set representation, every digit-column i would be nonzero and have any of the possible combinations $[b_i^{m-1}, \ldots, b_i^2, b_i^1, b_i^0]$, where $b_i^0 \in \{1, -1\}$, and $b_i^j \in \{0, b_i^0\}$ for $1 \leq j < m$. Since digits in the same column have the same sign, one only needs to precompute all the positive combinations $P_0 + u_1 P_1 + \ldots + u_{m-1} P_{m-1}$ with $u_j \in \{0, 1\}$, where P_j are the base points of the sub-scalars, and to leave the computation of the negative values to the evaluation stage. The cost of Alg. 2 is given by $(l-1)$ doublings and l additions during the evaluation stage using 2^{m-1} points, where $l = \lceil \frac{r}{m} \rceil + 1$. Naively, precomputation costs $2^{m-1} - 1$ additions.

So the total cost is given by $(l-1)$ doublings and $(l + 2^{m-1} - 1)$ additions. Compare this cost with the method presented in [20] based on a regular windowed recoding that requires $m \cdot (l-1)/(w-1)$ doublings and $m \cdot (l-1)/(w-1) + 2m - 1$ additions during the evaluation stage and m doublings with $m \cdot (2^{w-2} - 1)$ additions for precomputation using $m \cdot (2^{w-2}+1)$ points (naive approach without exploiting endomorphisms). If, for example, $m = 4$ and $w = 5$, the new method costs $(l-1)$ doublings and $(l+7)$ additions using 8 points, whereas the windowed method costs $(l+3)$ doublings and $(l+34)$ additions using 36 points. Thus, the new method improves performance while reduces dramatically the number of required precomputations.

Algorithm 2 Protected m-GLV Variable-Base Scalar Multiplication using the GLV-Based LSB-Set Representation.

Input: Base point P_0 of order r and $(m-1)$ points P_j for $1 \leq j < m$ corresponding to the endomorphisms, m scalars $k_j = (k^j_{t_j-1}, \ldots, k^j_0)_2$ for $0 \leq j < m$, $l = \lceil \frac{r}{m} \rceil + 1$ and $\max(t_j) = \lceil \frac{r}{m} \rceil$.
Output: kP.

Precomputation stage:
1: Compute $P[u] = P_0 + u_0 P_1 + \ldots + u_{m-2} P_{m-1}$ for all $0 \leq u < 2^{m-1}$, where $u = (u_{m-2}, \ldots, u_0)_2$.
Recoding stage:
2: even $= k_0 \bmod 2$
3: **if** even $= 0$ **then** $k_0 = k_0 - 1$
4: Pad each k_j with $(l - t_j)$ zeros to the left for $0 \leq j < m$ and convert them to the GLV-based LSB-set representation using Algorithm 1 s.t. $k_j = (b^j_{l-1}, \ldots, b^j_0)_{\text{LSB-set}}$. Set digit-columns $\mathbb{K}_i = [b^{m-1}_i, \ldots, b^2_i, b^1_i] \equiv |b^{m-1}_i 2^{m-2} + \ldots + b^2_i 2 + b^1_i|$ and digit-column signs $s_i = b^0_i$ for $0 \leq i \leq l-1$.
Evaluation stage:
5: $Q = s_{l-1} P[\mathbb{K}_{l-1}]$
6: **for** $i = l-2$ **to** 0 **do**
7: $\quad Q = 2Q$
8: $\quad Q = Q + s_i P[\mathbb{K}_i]$
9: **if** even $= 0$ **then** $Q = Q + P_0$
10: **return** Q

Since our recoding algorithm requires that the first sub-scalar k_0 be odd, in Step 3 of Algorithm 2 k_0 is subtracted by one if it is even. The correction is then performed at the end of the evaluation stage at Step 9. These computations, as well as the accesses to the precomputed table, should be performed in constant time to guarantee protection against timing attacks. For example, to perform Step 9 in our implementation we always carry out the computation $Q' = Q - P_0$. Then, we perform a linear pass over the points Q and Q' using conditional move instructions to transfer the correct value to the final destination.

Note that Algorithm 2 assumes a decomposed scalar as input. This is sufficient in some settings, in which randomly generated sub-scalars could be provided. However, in others settings, one requires to calculate the sub-scalars in a decomposition phase. This can be accomplished by using the lattice reduction described in [20] together with a standard decomposition based on Babai's rounding method. In practice, the computation is inexpensive since most operations can be performed offline. An additional issue is that some sub-scalars obtained from the decomposition can be negative. In this case, to work with Algorithm 1 and 2, negative sub-scalars should be converted to positive with the corresponding base point negated.

3.2 Fixed-Base Scalar Multiplication

Consider the same set up as at the beginning of §3.1. Similar to Feng et al. [7], Hedabou et al. [13] proposed the use of a signed odd-only representation with digits $\{1, -1\}$ by converting 1 to $1\bar{1}\ldots\bar{1}$. The main difference is that Hedabou et al.'s representation, referred to as signed all-bit-set (SAB-set), uses the transformation above to convert the whole scalar k to digits $\{1, -1\}$. As in [7], this representation is used in a comb-style execution for computing kP, when P is known in advance. The method is also resistant to SPA attacks since every "digit-column" is nonzero. In this case, the method requires to precompute all points $(b_{w-1}2^{(w-1)d} + \ldots + b_1 2^d + b_0)P$ for all $(b_{w-1}, \ldots, b_1, b_0) \in \{1, -1\}^w$. An important observation is that both

Feng et al. and Hedabou et al. used a simple version of the Lim-Lee's comb method that is restricted to only one table (see [17] for more details). Recently, Hamburg [12] precisely proposed to combine the original multi-table Lim-Lee's comb approach with Hedabou et al.'s SAB-set representation. In this work, we exploit this optimized approach to compute fixed-base scalar multiplication. The full details are next.

First, we propose an alternative odd-only recoding algorithm to obtain the SAB-set representation that is simple and runs in constant time, making it resilient to timing attacks. The details are shown in Algorithm 3. Note that the recoding can be implemented very efficiently using, for example, the computation $b_i = (k_{i+1} - 1) \,|\, 1$, where $|$ represents a logical OR operation.

Algorithm 3 Protected Odd-Only Recoding for the SAB-Set Representation.

Input: An odd positive integer $k = (k_{l-1}, \ldots, k_0)_2$.
Output: $k = (b_{l-1}, \ldots, b_0)_{\text{SAB-set}}$, where $b_i \in \{1, -1\}$ for $0 \leq i \leq l-1$.

1: $b_{l-1} = 1$
2: **for** $i = 0$ **to** $(l-2)$ **do**
3: $\quad b_i = 2k_{i+1} - 1$
4: **return** $(b_{l-1}, \ldots, b_0)_{\text{SAB-set}}$

Now we give the full algorithm (shown as Algorithm 4) for computing fixed-base scalar multiplication using the SAB-set representation, since this is not presented in [12]. Note that for dealing with an even scalar, we take advantage that our setting works on a subgroup of prime order r. Hence, $-k \pmod r = r - k$ gives an odd result. When computing fixed-base scalar multiplication, precomputation is assumed to be performed offline. Hence, the method requires $e - 1 = \lceil \frac{d}{v} \rceil - 1 = \lceil \frac{l}{w \cdot v} \rceil - 1$ doublings and $ev - 1 = v\lceil \frac{l}{w \cdot v} \rceil - 1$ additions, using $v \cdot 2^{w-1}$ precomputed points. The main loop of Algorithm 4 computes kP using the regular pattern one doubling, v additions, ..., one doubling, v additions and, hence, it provides the first step to protect against timing and SPA attacks. Moreover, the recoding, via Algorithm 3, is also protected. Similar to Alg. 2, to be fully resistant, conditional statements (e.g., Steps 3 and 9) and table accesses should also be carried out securely and in constant time.

Algorithm 4 Protected Fixed-Base Scalar Multiplication using the SAB-Set Comb Method.

Input: A point $P \in E(\mathbb{F}_q)$ of prime order r, a scalar $k = (k_{t-1}, \ldots, k_0)_2 \in [1, r-1]$, window width $w \geq 2$, table parameter $v \geq 1$, $d = \lceil t/w \rceil$ and $e = \lceil d/v \rceil$.
Output: kP.

Precomputation stage:
1: Compute $P[(u-1)/2][v'] = 2^{ev'}(u_0 + u_1 2^d + \ldots + u_{w-1} 2^{(w-1)d})P$ for all $u \in \{1, 3, 5, \ldots, 2^w - 1\}$ and $0 \leq v' < v$, where $u = (u_{w-1}, \ldots, u_0)_{\text{SAB-set}}$.
Recoding stage:
2: odd $= k \bmod 2$
3: **if** odd $= 0$ **then** $k = r - k$
4: Pad k with $(dw - t)$ zeros to the left and convert it to the SAB-set representation using Algorithm 3 s.t. $k = (b_{l-1}, \ldots, b_0)_{\text{SAB-set}}$, where $l = dw$. Set $k = K^{w-1} \,||\, \ldots \,||\, K^1 \,||\, K^0$, where each $K^{w'}$ consists of v strings of e digits each. Let the v'-th string in a given $K^{w'}$ be denoted by $K^{w'}_{v'}$, and the e'-th digit in a given $K^{w'}_{v'}$ be denoted by $K^{w'}_{v',e'}$, s.t. $K^{w'}_{v',e'} = b_{dw'+ev'+e'}$. Set digit-columns $\mathbb{K}_{v',e'} = [K^{w-1}_{v',e'}, \ldots, K^1_{v',e'}, K^0_{v',e'}] \equiv |K^{w-1}_{v',e'} 2^{w-1} + \ldots + K^1_{v',e'} 2 + K^0_{v',e'}|$, and digit-column signs $s_{v',e'}$ s.t. $s_{v',e'} = 1$ if $K^{w-1}_{v',e'} 2^{w-1} + \ldots + K^1_{v',e'} 2 + K^0_{v',e'} \geq 0$, else $s_{v',e'} = -1$.
Evaluation stage:
5: $Q = s_{0,e-1} P[(\mathbb{K}_{0,e-1} - 1)/2][0] + s_{1,e-1} P[(\mathbb{K}_{1,e-1} - 1)/2][1] + \ldots + s_{v-1,e-1} P[(\mathbb{K}_{v-1,e-1} - 1)/2][v-1]$
6: **for** $i = e - 2$ **to** 0 **do**
7: $\quad Q = 2Q$
8: $\quad Q = Q + s_{0,i} P[(\mathbb{K}_{0,i} - 1)/2][0] + s_{1,i} P[(\mathbb{K}_{1,i} - 1)/2][1] + \ldots + s_{v-1,i} P[(\mathbb{K}_{v-1,i} - 1)/2][v-1]$
9: **if** odd $= 0$ **then** $Q = -Q$
10: **return** Q

We note that for the GLV-setting Algorithm 4 does not exploit endomorphisms. It is an open problem to find a scalar multiplication method that exploits endomorphisms efficiently in the fixed-base case.

3.3 Double Scalar Multiplication with Fixed and Variable Bases

This computation can be efficiently computed using wNAF with interleaving [11, 22]. Since precomputation for the fixed base is performed offline, one may arbitrarily increase the window size for this case, only taking into consideration any memory restriction. When using the m-dimensional GLV, the two scalars can be split in two sets of m sub-scalars with maximum bitlength $l = \lceil r/m \rceil$ each. The cost is given by $m \cdot (\frac{l}{w_1+1}) - 1$ additions, $m \cdot (\frac{l}{w_2+1})$ mixed additions and $(l-1)$ doublings using $m \cdot (2^{w_1-2} + 2^{w_2-2})$ precomputed points, where w_1 is the window size for the variable base and w_2 is the window size for the fixed base. The cost of *online* precomputation is naivily m doublings and $m \cdot (2^{w_1} - 1)$ additions, without exploiting endomorphisms. Note that it is possible to choose different window sizes for each sub-scalar, giving more flexibility in the selection of the optimal number of precomputations. This is proven to be useful in the cost analysis in §6.

Since this scenario is typical for signature verification, the cost of m-GLV decomposition of the two scalars and the cost of recoding to wNAF representation should be added. Also, one could employ unprotected versions of field multiplication, squaring and inversion, which are somewhat faster than protected ones.

4 Field and Quadratic Extension Field Arithmetic

Next, we describe implementation details and optimized algorithms for field and extension field operations.

4.1 Field Arithmetic

For field inversion, we use the modular exponentiation $a^{p-2} \pmod p \equiv a^{-1}$ using a fixed and short addition chain. This method is simple to implement and is naturally protected against timing attacks.

In the case of a pseudo-Mersenne prime of the form $p = 2^m - c$, with c small, field multiplication can be efficiently performed by computing an integer multiplication followed by a modular reduction exploiting the special form of the prime. This separation of operations also enables the use of lazy reduction in the extension field arithmetic. For x64, integer multiplication is implemented in product scanning form (a.k.a Comba's method), mainly exploiting the powerful 64-bit unsigned multiplier instruction. Let $0 \leq a, b < 2^{m+1}$. To exploit the extra room of one bit in our targeted prime $2^{127}-5997$, we first compute $M = a \cdot b = 2^{m+1}M_H + M_L$ followed by the reduction step $R = M_L + 2c\,M_H \leq 2^{m+1}(2c+1) - 2$. Then, given $R = 2^m R_H + R_L$, we compute $R_L + c\,R_H \pmod p$, where $R_L, c\,R_H < 2^m$. This final operation can be efficiently carried out by employing the modular addition proposed by Bos et al. [5] to get the final result in the range $[0, p-1]$. Note that the computation of field multiplication above naturally accepts inputs in unreduced form without incurring in extra costs, enabling the use of additions without correction or operations with incomplete reduction (see below for more details). We follow a similar procedure for computing field squaring. For ARM, we implement the integer multiplication using the schoolbook method. In this case, and also for modular reduction, we extensively exploit the parallelism of ARM and NEON instructions. The details are discussed in Section 4.3.

Let $0 \leq a, b < 2^m - c$. Field subtraction is computed as $(a - b) + \mathit{borrow} \cdot 2^m - \mathit{borrow} \cdot c$, where $\mathit{borrow} = 0$ if $a \geq b$, otherwise $\mathit{borrow} = 1$. Notice that in practice the addition with $\mathit{borrow} \cdot 2^m$ can be efficiently implemented by clearing the $(m+1)$-th bit of $a - b$.

Incomplete Reduction. Similar to [19], we exploit the form of the pseudo-Mersenne prime in combination with the incomplete reduction technique to speedup computations. We also mix incompletely reduced and completely reduced operands in novel ways.

Let $0 \leq a < 2^m - c$ and $0 \leq b < 2^m$. Field addition with incomplete reduction is computed as $(a + b) - \mathit{carry} \cdot 2^m + \mathit{carry} \cdot c$, where $\mathit{carry} = 0$ if $a + b < 2^m$, otherwise $\mathit{carry} = 1$. Again, in practice the subtraction with $\mathit{carry} \cdot 2^m$ can be efficiently implemented by clearing the $(m+1)$-th bit of $a + b$. The result is correct modulo p but falls in the range $[0, 2^m - 1]$. Thus, this addition operation with incomplete reduction works with both operands in completely reduced form or with one operand in completely reduced form and one in incompletely reduced form. A similar observation applies to subtraction. Consider two operands a, b such that $0 \leq a < 2^m$ and $0 \leq b < 2^m - c$. Then, the standard field subtraction $(a - b) \bmod (2^m - c)$ described above will

produce an incompletely reduced result in the range $[0, 2^m - 1]$, since $a - b$ with $borrow = 0$ produces a result in the range $[0, 2^m - 1]$ and $a - b$ with $borrow = 1$ produces a result in the range $[-2^m + c + 1, -1]$, which is then fully reduced by adding $2^m - c$. Thus, performance can be improved by using incomplete reduction in an addition preceding a subtraction. For example, this technique is exploited in the point doubling computation (see Steps 7-8 of Algorithm 10). Note that, in contrast to addition, only the first operand is allowed to be in incompletely reduced form for subtraction.

To guarantee correctness in our software, and following the previous description, incompletely reduced results are always fed to one of the following: one of the operands of an incompletely reduced addition, the first operand of a field subtraction, a field multiplication or squaring (which ultimately produces a completely reduced output), or a field addition without correction preceding a field multiplication or squaring.

In the targeted setting, there are only a limited number of spots in the curve arithmetic in which incompletely reduced numbers cannot be efficiently exploited. For these few cases, we require a standard field addition. We use the efficient implementation proposed by Bos et al. [5]. Again, let $0 \leq a, b < 2^m - c$. Field addition is then computed as $((a + c) + b) - carry \cdot 2^m - (1 - carry) \cdot c$, where $carry = 0$ if $a + b + c < 2^m$, otherwise $carry = 1$. Similar to previous cases, the subtraction with $carry \cdot 2^m$ can be efficiently carried out by clearing the $(m+1)$-th bit in $(a+c)+b$. As we discussed it, this efficient computation is also advantageously exploited in the modular reduction for multiplication and squaring.

4.2 Quadratic Extension Field Arithmetic

For the remainder, we use the following notation: (i) I, M, S, A and R represent field inversion, field multiplication, field squaring, field addition and modular reduction, respectively, (ii) M_i and A_i represent integer multiplication and integer addition, respectively, and (iii) i, m, s, a and r represent analogous operations over $\mathbb{F}_{p^2}$. When representing registers in algorithms, capital letters are used to allocate operands with "double precision" (in our case, 256 bits). For simplification purposes, in the operation counting an integer operation with double-precision is considered equivalent to two integer operations with single precision. We assume that addition, subtraction, multiplication by two and negation have roughly the same cost.

Let $a = a_0 + a_1 i \in \mathbb{F}_{p^2}$ and $b = b_0 + b_1 i \in \mathbb{F}_{p^2}$. Inversion over $\mathbb{F}_{p^2}$ is computed as $a^{-1} = (a_0 - a_1 i)/(a_0^2 + a_1^2)$. Addition and subtraction over $\mathbb{F}_{p^2}$ consist in computing $(a_0 + b_0) + (a_1 + b_1)i$ and $(a_0 - b_0) + (a_1 - b_1)i$, respectively. We compute multiplication over $\mathbb{F}_{p^2}$ using the Karatsuba method. In this case, we fully exploit lazy reduction and the room of one bit that is gained by using a prime of 127 bits. The details for the x64 implementation are shown in Algorithm 5. Remarkably, note that only the subtraction in Step 3 requires a correction to produce a positive result. No other addition or subtraction requires correction to positive or to modulo p. That is, $\times$, $+$ and $-$ represent operations over the integers. In addition, the algorithm accepts inputs in completely or incompletely reduced form and always produces a result in completely reduced form. Optionally, one may "delay" the computation of the final modular reductions (by setting $reduction = FALSE$ in Alg. 5) if lazy reduction could be exploited in the curve arithmetic. This has been proven to be useful to formulas for the Weierstrass form [1], but unfortunately the technique cannot be advantageously exploited in the most efficient formulas for Twisted Edwards (in this case, one should set $reduction = TRUE$). Squaring over $\mathbb{F}_{p^2}$ is computed using the complex method. The details for the x64 implementation are shown in Algorithm 6. In this case, all the additions are computed as integer operations since, again, results can be let to grow up to 128 bits, letting subsequent multiplications take care of the reduction step.

4.3 Field Arithmetic on ARM: Efficient Interleaving of ARM and NEON Instructions

The ARM architecture comes equipped with 16 32-bit registers and an instruction set including 32-bit operations which in most cases can be executed in one cycle. To boost performance in some applications, the targeted ARM platforms include the powerful vector set of instructions NEON. This consists of a 128-bit Single Instruction Multiple Data (SIMD) engine that includes 16 128-bit registers, which can be seen as 32 64-bit registers or as 16 128-bit registers.

In [25], the authors show how to take advantage of NEON instructions to perform independent multiplications inside operations over $\mathbb{F}_{p^2}$. In the following, we give a step further and show how to exploit the

Algorithm 5 Multiplication in $\mathbb{F}_{p^2}$ with reduction ($m = 3M_i + 9A_i + 2R$) and without reduction ($m_u = 3M_i + 9A_i$), using completely or incompletely reduced inputs (x64 platform).

Input: $a = (a_0 + a_1 i)$ and $b = (b_0 + b_1 i) \in \mathbb{F}_{p^2}$, where $0 \leq a_0, a_1, b_0, b_1 \leq 2^{127} - 1, p = 2^{127} - c, c$ small.
Output: $a \cdot b \in \mathbb{F}_{p^2}$.

1: $T_0 \leftarrow a_0 \times b_0$	$[0, 2^{254} >$
2: $T_1 \leftarrow a_1 \times b_1$	$[0, 2^{254} >$
3: $C_0 \leftarrow T_0 - T_1$	$< -2^{254}, 2^{254} >$
4: **if** $C_0 < 0$, **then** $C_0 \leftarrow C_0 + 2^{128} \cdot p$	$[0, 2^{255} >$
5: **if** *reduction*$= TRUE$, **then** $c_0 \leftarrow C_0 \bmod p$	$[0, p >$
6: $t_0 \leftarrow a_0 + a_1$	$[0, 2^{128} >$
7: $t_1 \leftarrow b_0 + b_1$	$[0, 2^{128} >$
8: $T_2 \leftarrow t_0 \times t_1$	$[0, 2^{256} >$
9: $T_2 \leftarrow T_2 - T_0$	$[0, 2^{256} >$
10: $C_1 \leftarrow T_2 - T_1$	$[0, 2^{256} >$
11: **if** *reduction*$= TRUE$, **then** $c_1 \leftarrow C_1 \bmod p$	$[0, p >$
12: **return** if *reduction*$= TRUE$ then $a \cdot b = (c_0 + c_1 i)$, else $a \cdot b = (C_0 + C_1 i)$.	

Algorithm 6 Squaring in $\mathbb{F}_{p^2}(s = 2M + 1A + 2A_i)$, using completely reduced inputs (x64 platform).

Input: $a = (a_0 + a_1 i) \in \mathbb{F}_{p^2}$, where $0 \leq a_0, a_1 \leq p - 1, p = 2^{127} - c, c$ small.
Output: $a^2 \in \mathbb{F}_{p^2}$.

1: $t_0 \leftarrow a_0 + a_1$	$[0, 2^{128} >$
2: $t_1 \leftarrow a_0 - a_1 \bmod p$	$[0, p >$
3: $c_0 \leftarrow t_0 \times t_1 \bmod p$	$[0, p >$
4: $t_0 \leftarrow a_0 + a_0$	$[0, 2^{128} >$
5: $c_1 \leftarrow t_0 \times a_1 \bmod p$	$[0, p >$
6: **return** $a^2 = (c_0 + c_1 i)$.	

increasingly efficient capacity of modern ARM processors for executing ARM and NEON instructions "simultaneously" to implement field operations. In other words, we exploit the fact that when ARM code produces a data hazard in the pipeline, the NEON unit may be ready to execute vector instructions, and viceversa. Note that loading/storing values from ARM to NEON registers still remains relatively expensive, so in order to achieve an effective performance improvement, one should carefully interleave *independent* operations while minimizing the loads and stores from one unit to the other. Hence, operations such as multiplication, squaring and reduction over $\mathbb{F}_{p^2}$ are particularly friendly to this technique, given the availability of internal independent multiplications in their formulas. Thus, using this approach, we implemented:

- a double integer multiplier (`double_mul_neonarm`) detailed in Algorithm 8, which interleaves a single 128-bit multiplication using NEON and a single 128-bit multiplication using ARM,
- a triple integer multiplier (`triple_mul_neonarm`) detailed in Algorithm 7, which interleaves two single 128-bit multiplication using NEON and one single 128-bit multiplication using ARM, and
- a double reduction algorithm, denoted by `double_red_neonarm` and detailed in Algorithm 9, that interleaves a single modular reduction using NEON and a single modular reduction using ARM.

Note that integer multiplication is implemented using the schoolbook method, which requires one multiplication, two additions, one shift and one bit-wise AND operation per iteration. These operations were implemented using efficient fused instructions such as UMLAL, UMAAL, VMLAL and VSRA[18], which add the result of a multiply or shift operation to the destination register in one single operation, reducing code size.

Algorithm 7 Triple 128-bit integer product with ARM and NEON interleaved (`triple_mul_neonarm`).

Input: $a = \{a_i\}, b = \{b_i\}, c = \{c_i\}, d = \{d_i\}, e = \{e_i\}, f = \{f_i\}$, $i \in \{0, \ldots, 3\}$.
Output: $(F, G, H) \leftarrow (a \times b, c \times d, e \times f)$.

$(F, G, H) \leftarrow (0, 0, 0)$
for $i = 0$ **to** 3 **do**
  $(C_0, C_1, C_2) \leftarrow (0, 0, 0)$
  for $j = 0$ **to** 3 **do**
    $(C_0, F_{i+j}, C_1, G_{i+j}) \leftarrow (F_{i+j} + a_j b_i + C_0, G_{i+j} + c_j d_i + C_1)$ {done by NEON}
  for $j = 0$ **to** 3 **do**
    $(C_2, H_{i+j}) \leftarrow H_{i+j} + e_j f_i + C_2$ {done by ARM}
  $(F_{i+4}, G_{i+4}, H_{i+4}) \leftarrow (C_0, C_1, C_2)$
return (F, G, H)

Algorithm 8 Double 128-bit integer product with ARM and NEON interleaved (`double_mul_neonarm`).

Input: $a = \{a_i\}, b = \{b_i\}, c = \{c_i\}, d = \{d_i\}$, $i \in \{0, \ldots, 3\}$.
Output: $(F, G) \leftarrow (a \times b, c \times d)$.

$(F, G) \leftarrow (0, 0)$
for $i = 0$ **to** 1 **do**
  $(C_0, C_1, C_2) \leftarrow (0, 0, 0)$
  for $j = 0$ **to** 3 **do**
    $(C_0, F_{i+j}, C_1, F_{i+j+2}) \leftarrow (F_{i+j} + a_i b_j + C_0, F_{i+j+2} + a_{i+2} b_j + C_1)$ {done by NEON}
  for $j = 0$ **to** 3 **do**
    $(C_2, G_{i+j}) \leftarrow G_{i+j} + c_j d_i + C_2$ {done by ARM}
  $(F_{i+4}, F_{i+6}, G_{i+4}) \leftarrow (F_{i+4} + C_0, C_1, C_2)$
for $i = 2$ **to** 3 **do**
  for $j = 0$ **to** 3 **do**
    $(C_2, G_{i+j}) \leftarrow G_{i+j} + c_j d_i + C_2$ {done by ARM}
  $G_{i+4} \leftarrow C_2$
return (F, G)

Algorithm 9 Double modular reduction with ARM and NEON interleaved (`double_red_neonarm`).

Input: A prime $p = 2^{127} - c$, $a = \{a_i\}, b = \{b_i\}$, $i \in \{0, \ldots, 7\}$.
Output: $(F, G) \leftarrow (a \mod p,\ b \mod p)$.

$(F_i, G_i) \leftarrow (a_i, b_i)_{i \in \{0,\ldots,3\}}$
$(C_0, C_1, C_2) \leftarrow (0, 0, 0)$
for $j = 0$ **to** 1 **do**
  $(C_0, F_j, C_1, F_{j+2}) \leftarrow (F_j + a_{j+4} c + C_0, F_{j+2} + a_{j+6} c + C_1)$ {done by NEON}
for $j = 0$ **to** 3 **do**
  $(C_2, G_j) \leftarrow G_j + b_{j+4} c + C_2$ {done by ARM}
$(F_2, F_4, G_4) \leftarrow (F_2 + C_0, C_1, C_2)$
$(F_0, G_0) \leftarrow (F_4 c + F_0, G_4 c + G_0)$
return (F, G)

To validate the efficiency of our approach, we compared the interleaved algorithms above with standard implementations using NEON or ARM. In all the cases, we observed a reduction of costs in favor of our novel interleaved ARM/NEON implementations.

`Triple_mul_neonarm` is nicely adapted to the computation of multiplication over $\mathbb{F}_{p^2}$, since this operation requires three integer multiplications of 128 bits (Steps 1, 2 and 8 of Algorithm 5). For the case of squaring over $\mathbb{F}_{p^2}$, we use `double_mul_neonarm` to compute the two independent integer multiplications (Steps 3 and 5 of Algorithm 6). Finally, for each case we can efficiently use a `double_red_neonarm`. The final algorithms for ARM are shown as Algorithms 12 and 13 in Appendix B.

5 Point Arithmetic

In this section, we describe implementation details and our optimized formulas for the point arithmetic. We use as basis the most efficient set of formulas proposed by Hisil et al. [14], corresponding to the case $a = -1$, that uses a combination of homogeneous projective coordinates $(X : Y : Z)$ and extended homogeneous coordinates of the form $(X : Y : Z : T)$, where $T = XY/Z$.

The basic algorithms for computing point doubling and addition are shown in Algorithms 10 and 11, respectively. In these algorithms, we extensively exploit incomplete reduction, following the details given in Section 4 (operations with incomplete reduction are represented with $\oplus, \ominus$). To ease coupling of doubling and addition in the main loop of the scalar multiplication computation, we make use of Hamburg's "extensible" strategy and output values $\{T_a, T_b\}$, where $T = T_a \cdot T_b$, at every point operation, so that a subsequent operation may compute coordinate T if required. Note that the cost of doubling is given by $4m+3s+5a$. We do not apply the usual transformation $2XY = (X+Y)^2-(X^2+Y^2)$ because in our case it is faster to compute one multiplication and one incomplete addition than one squaring, one subtraction and one addition. In the setting of variable-base scalar multiplication (see Alg. 2), the main loop of the evaluation stage consists of a doubling-addition computation, which corresponds to the successive execution of Algorithms 10 and 11. For this case, precomputed points are more efficiently represented as $(X+Y, Y-X, 2Z, 2T)$ (corresponding to setting $EXT\_COORD = TRUE$ in Alg. 11), so the cost of addition is given by $8m + 6a$. In the fixed-base scenario, the main loop of the evaluation stage consists of one doubling and v mixed additions. In this case, we consider two possibilities: representing precomputations as (x, y) or as $(x + y, y - x, 1, 2t)$, where $t = xy$. The latter case (also corresponding to setting $EXT\_COORD = TRUE$, but with $Z = 1$) allows saving four additions and one multiplication per iteration (Steps 2, 8 and 11 of Alg. 11), but increases the memory requirements to store the additional coordinate. Hence, each option ends up being optimal for certain storage values. We evaluate these options in Section 6. For the case (x, y), mixed addition costs $8m + 10a$ and, for the case $(x+y, y-x, 1, 2t)$, mixed addition costs $7m+7a$. In the variable/fixed-base double scalar scenario, precomputed points corresponding to the variable base are stored as $(X + Y, Y - X, 2Z, 2T)$ and, thus, addition with these points costs $8m+6a$; whereas points corresponding to the fixed base can again be repre-

Algorithm 10 Twisted Edwards point doubling over $\mathbb{F}_{p^2}$ (DBL $= 4m + 3s + 5a$).

Input: $P = (X_1, Y_1, Z_1)$.
Output: $2P = (X_2, Y_2, Z_2)$ and $\{T_a, T_b\}$ such that $T_2 = T_a \cdot T_b$.

1: $T_a \leftarrow X_1^2$	(X_1^2)
2: $t_1 \leftarrow Y_1^2$	(Y_1^2)
3: $T_b \leftarrow T_a \oplus t_1$	$(X_1^2 + Y_1^2)$
4: $T_a \leftarrow t_1 - T_a$	$(Y_1^2 - X_1^2)$
5: $Y_2 \leftarrow T_b \times T_a$	$(Y_2 = (X_1^2 + Y_1^2)(Y_1^2 - X_1^2))$
6: $t_1 \leftarrow Z_1^2$	(Z_1^2)
7: $t_1 \leftarrow t_1 \oplus t_1$	$(2Z_1^2)$
8: $t_1 \leftarrow t_1 \ominus T_a$	$(2Z_1^2 - (Y_1^2 - X_1^2))$
9: $Z_2 \leftarrow T_a \times t_1$	$(Z_2 = (Y_1^2 - X_1^2)[2Z_1^2 - (Y_1^2 - X_1^2)])$
10: $T_a \leftarrow X_1 \oplus X_1$	$(2X_1)$
11: $T_a \leftarrow T_a \times Y_1$	$(2X_1Y_1)$
12: $X_2 \leftarrow T_a \times t_1$	$(X_2 = 2X_1Y_1[2Z_1^2 - (Y_1^2 - X_1^2)])$
13: **return** $2P = (X_2, Y_2, Z_2)$ and $\{T_a, T_b\}$ such that $T_2 = T_a \cdot T_b$.	

sented as (x, y) or as $(x+y, y-x, 1, 2t)$, following the same trade-offs and costs for mixed addition discussed above. These options are also evaluated in Section 6.

Algorithm 11 Twisted Edwards point addition over $\mathbb{F}_{p^2}$ (ADD $= 8m+6a$, mADD $= 7m+7a$ or $8m+10a$).

Input: $P = (X_1, Y_1, Z_1)$ and $\{T_a, T_b\}$ such that $T_1 = T_a \cdot T_b$. If $EXT\_COORD = FALSE$ then $Q = (x_2, y_2)$, else $Q = (X_2 + Y_2, Y_2 - X_2, 2Z_2, 2T_2)$.
Output: $P + Q = (X_3, Y_3, Z_3)$ and $\{T_a, T_b\}$ such that $T_3 = T_a \cdot T_b$.

1: $T_1 \leftarrow T_a \times T_b$	(T_1)
2: **if** $EXT\_COORD = FALSE$ **then** $T_2 = x_2 \oplus x_2$, $T_2 = T_2 \times y_2$	$(2T_2)$
3: $t_1 \leftarrow T_2 \times Z_1$	$(2T_2Z_1)$
4: **if** $Z_2 = 1$ **then** $t_2 \leftarrow T_1 \oplus T_1$ **else** $t_2 \leftarrow T_1 \times Z_2$	$(2T_1Z_2)$
5: $T_a \leftarrow t_2 - t_1$	$(T_a = \alpha = 2T_1Z_2 - 2T_2Z_1)$
6: $T_b \leftarrow t_1 \oplus t_2$	$(T_b = \theta = 2T_1Z_2 + 2T_2Z_1)$
7: $t_2 \leftarrow X_1 \oplus Y_1$	$(X_1 + Y_1)$
8: **if** $EXT\_COORD = TRUE$ **then** $Y_3 = Y_2$, **else** $Y_3 = y_2 - x_2$	$(Y_2 - X_2)$
9: $t_2 \leftarrow Y_3 \times t_2$	$(X_1 + Y_1)(Y_2 - X_2)$
10: $t_1 \leftarrow Y_1 - X_1$	$(Y_1 - X_1)$
11: **if** $EXT\_COORD = TRUE$ **then** $X_3 = X_2$, **else** $X_3 = x_2 \oplus y_2$	$(X_2 + Y_2)$
12: $t_1 \leftarrow X_3 \times t_1$	$(X_2 + Y_2)(Y_1 - X_1)$
13: $Z_3 \leftarrow t_2 - t_1$	$\beta = (X_1 + Y_1)(Y_2 - X_2) - (X_2 + Y_2)(Y_1 - X_1)$
14: $t_1 \leftarrow t_1 \oplus t_2$	$\omega = (X_1 + Y_1)(Y_2 - X_2) + (X_2 + Y_2)(Y_1 - X_1)$
15: $X_3 \leftarrow T_b \times Z_3$	$(X_3 = \beta\theta)$
16: $Z_3 \leftarrow t_1 \times Z_3$	$(Z_3 = \beta\omega)$
17: $Y_3 \leftarrow T_a \times t_1$	$(Y_3 = \alpha\omega)$
18: **return** $P + Q = (X_3, Y_3, Z_3)$ and $\{T_a, T_b\}$ such that $T_3 = T_a \cdot T_b$.	

6 Performance Analysis and Experimental Results

In this section, we carry out the performance analysis of the different methods discussed in this work and present benchmark results of our constant-time implementations of curve `Ted127-glv4` on x64 and ARM platforms. For our experiments, we targeted a 3.4GHz Intel Core i7-2600 Sandy Bridge processor and a 3.4GHz Intel Core i7-3770 Ivy Bridge processor, from the Intel family, and a Samsung Galaxy Note with a 1.4GHz Exynos 4 Cortex-A9 processor and an Arndale Board with a 1.7GHz Exynos 5 Cortex-A15 processor, from the ARM family, both equipped with the NEON vector unit. The x64 implementation was compiled with Microsoft Visual Studio 2012 and ran on 64-bit Windows (Microsoft Windows 8 OS). In our experiments, we turned off hype threading and Intel's Turbo Boost; and we averaged the cost of 10^4 operations which were measured with the timestamp counter instruction `rdtsc`. The ARM implementation was developed and compiled with the Android NDK (ndk8d) toolkit. In this case, we averaged the cost of 10^4 operations which were measured with the `clock_gettime()` function and scaled to clock cycles using the processor frequency.

First, in Table 1 we present timings for all the fundamental operations that are necessary to implement the different cases of scalar multiplication. Implementation details for quadratic extension field operations and point operations over $\mathbb{F}_{p^2}$ can be found in Section 4 and 5, respectively. "IR" stands for incomplete reduction and "extended" represents the use of the extended coordinates $(X+Y, Y-X, 2Z, 2T)$ to represent precomputed points.

Next, we analyze the different scalar multiplication scenarios on curve `Ted127-glv4`.

Variable-Base Scenario. Based on Algorithm 2, scalar multiplication on curve `Ted127-glv4` involves the computation of one Φ endomorphism, 2 Ψ endomorphisms, 3 additions and 4 mixed additions in the precomputation stage; 63 doublings, 63 additions, one mixed addition and 64 protected table lookups in the evaluation stage; and one inversion and 2 multiplications over $\mathbb{F}_{p^2}$ for converting the final result to affine:

Table 1. Cost (in cycles) of basic operations on curve `Ted127-glv4`.

Operation		ARM Cortex-A9	ARM Cortex-A15	Intel Sandy Bridge	Intel Ivy Bridge
$\mathbb{F}_{p^2}$	ADD with IR	20	19	12	12
	ADD	39	37	15	15
	SUB	39	37	12	12
	SQR	223	141	59	56
	MUL	339	185	78	75
	INV	13,390	9,675	6,060	5,890
ECC	DBL	2,202	1,295	545	525
	ADD	3,098	1,831	690	665
	mADD ($Z_1 = 1$)	2,943	1,687	622	606
	ϕ endomorphism ($Z_1 = 1$)	3,118	1,724	745	712
	ψ endomorphism ($Z_1 = 1$)	1,644	983	125	119
Misc	8-point LUT (extended)	291	179	83	79
	GLV-based LSB-set recoding	1,236	873	482	482
	4-GLV decomposition	756	430	305	290

$\text{COST}_{variable\_kP} = 1i + 833m + 191s + 769a + 64LUT8 + 4M + 9A.$

This operation count does not take into account other additional computations, such as the recoding to the GLV-based LSB-set representation, which are relatively inexpensive. Also, for settings in which a full scalar is provided, one needs to add the cost of decomposition to 4-GLV. Compared to [20], the optimized method for variable-base scalar multiplication introduces a reduction in about 181 multiplications, 26 squarings and 228 additions over $\mathbb{F}_{p^2}$. Additionally, it only requires 8 precomputed points which involve 64 protected table lookups over 8 points (denoted by $LUT8$) during scalar multiplication, whereas the method in [20] requires 36 precomputed points which involve 68 protected table lookups over 9 points.

Fixed-Base Scenario. In this case, we analyze costs when using the SAB-set comb method (Algorithm 4) with 32, 64, 128, 256 and 512 precomputed points. Recalling Section 3.2, and given $d = \lceil 251/w \rceil$ and $l = dw$, the method costs $(\frac{l}{w \cdot v} - 1)$ doublings and $(\frac{l}{w} - 1)$ additions (assuming that $v|d$) using $v \cdot 2^{w-1}$ points. Again, there are two options: storing points as (x, y) (affine coordinates) or as $(x + y, y - x, 2t)$ ("extended" affine coordinates). We show in Table 3, Appendix C, the costs in terms of multiplications over $\mathbb{F}_{p^2}$ per bit for curve `Ted127-glv4`. Best results for a given memory requirement are highlighted. And, as can be seen, on the target platforms each precomputation representation is optimal for determined storage values. The latter is true in most cases; however, for high memory values, the results are mixed and depend on the platform.

Variable/Fixed-Base Double Scalar Scenario. In this case, we analyze the cost of `Ted127-glv4` for different values of $w_{2,j}$ (corresponding to each of the j sub-scalars of the fixed base, where $0 \leq j < 4$), with the optimal value $w_1 = 4$ for the sub-scalars of the variable base. This value for w_1 was determined during experimentation on the targeted platforms. Let $l = 63$ be the maximum bitlength of the eight sub-scalars. The computation approximately involves 62 doublings, 48 additions and $(\sum_{j=0}^{3} \frac{63}{w_{2,j}+1}) + 3$ mixed additions in the evaluation stage using 16 "ephemeral" precomputed points and $\sum_{j=0}^{3} 2^{w_{2,j}-2}$ "permanent" precomputed points; 2 doublings, 6 additions, 8 Ψ endomorphisms and one Φ endomorphism in the online precomputation stage; and one inversion with 2 multiplications over $\mathbb{F}_{p^2}$ to convert the final result to affine. Again, we examine storing points as (x, y) or as $(x + y, y - x, 2t)$. We show in Table 4, Appendix D, the costs in terms of multiplications over $\mathbb{F}_{p^2}$ per bit for curve `Ted127-glv4`. In this case, extended coordinates offer a higher performance in all cases. This is mainly due to the reduced cost for extracting points from the precomputed table, which is not required to be performed in constant time in

this scenario.

Finally, in Table 2 we summarize benchmark results for all the core scalar multiplication operations: variable-base, fixed base and variable/fixed-base double scalar scenario. The results for the representative variable-base scenario set a new speed record for protected elliptic curve scalar multiplication on x64 and ARM processors. In comparison with the previously fastest x64 implementation by Longa and Sica [20], which runs in 137,000 cycles, the presented result injects a cost reduction of 28% on a Sandy Bridge machine. Likewise, in comparison with the state-of-the-art genus 2 implementation by Bos et al. [5], which runs in 117,000 cycles, our result is 20% faster on an Ivy Bridge machine. It is also 14% faster than the very recent protected implementation by Oliveira et al. [23] based on a binary GLS curve using the GLV method, which runs in 114,000 cycles on a Sandy Bridge machine. Moreover, our results also demonstrate that the proposed techniques bring a dramatic reduction in the overhead for protecting against timing attacks. An unprotected version of our implementation computes a scalar multiplication in 87,000 cycles on the Sandy Bridge processor, which is only 11% faster than our protected version. In the case of ARM, our implementation of variable-base scalar multiplication on curve `Ted127-glv4` is 27% faster than Bernstein and Schwabe's `curve25519` implementation, which runs in 568,000 on a Cortex-A9 [4].

We achieve similar results in the fixed-base and double scalar scenarios. For instance:

- Hamburg [12] computes key generation (dominated by a fixed-base scalar multiplication) in 60K cycles on a Sandy Bridge and 254K cycles on a Cortex-A9 (without NEON) using a table of size 7.5KB. Using only 6KB, our software runs a fixed-base scalar multiplication in 54K cycles and 204K cycles, respectively.
- Hamburg [12] computes signature verification (dominated by a double scalar multiplication) in 169K cycles on a Sandy Bridge and 618K cycles on a Cortex-A9 (without NEON) using a table of size 3KB. Our software runs a double scalar multiplication in only 123K cycles and 495K cycles, respectively, using the same table size.

Table 2. Cost (in 10^3 cycles) of core scalar multiplication operations on curve `Ted127-glv4` with full protection against timing-type side-channel attacks at approximately 128-bit security level. Results are approximated to the nearest 1000 cycles.

Scalar Multiplication		ARM	ARM	Intel	Intel
Type	Parameters	Cortex-A9	Cortex-A15	Sandy Bridge	Ivy Bridge
kP, variable base	8 points (computed online)	417	244	98	94
kP, fixed base	$v = 4, w = 5$, 64 points, 6KB, extended	204	116	54	53
	$v = 4, w = 6$, 128 points, 12KB, extended	181	106	50	49
	$v = 8, w = 6$, 256 points, 24KB, extended	172	100	48	46
$kP + lQ$	$w_2 = 3$, 8 points, 768 bytes, extended	560	321	136	130
	$w_2 = 5$, 32 points, 3KB, extended	495	285	123	118
	$w_2 = 7$, 128 points, 12KB, extended	463	266	116	111

Acknowledgements: We would like to thank Francisco Rodríguez-Henríquez for his useful comments and for giving us access to the Arndale board for the development of the ARM implementation.

References

1. D.F. Aranha, K. Karabina, P. Longa, C. Gebotys, and J. López. Faster explicit formulas for computing pairings over ordinary curves. In *Advances in Cryptology - EUROCRYPT*, volume 6632, pages 48–68. Springer, 2011.
2. D. Bernstein. Cache-timing attacks on AES. 2005. `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.
3. D. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters. Twisted Edwards curves. In S. Vaudenay, editor, *Proceedings of Africacrypt 2008*, volume 5023 of *LNCS*, pages 389–405. Springer, 2008.

4. D. Bernstein and T. Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems, accessed on February 02, 2013. `http://bench.cr.yp.to/results-dh.html`.
5. J.W. Bos, C. Costello, H. Hisil, and K. Lauter. Fast cryptography in genus two (two is greater than one). In *Advances in Crytology - EUROCRYPT (to appear)*, 2013. Also in Cryptology ePrint Archive, Report 2012/670, `http://eprint.iacr.org/2012/670`.
6. D. Brumley and D. Boneh. Remote timing attacks are practical. In S. Mangard and F.-X. Standaert, editors, *Proceedings of the 12th USENIX Security Symposium*, volume 6225 of *LNCS*, pages 80–94. Springer, 2003.
7. M. Feng, B.B. Zhu, M. Xu, and S. Li. Efficient comb elliptic curve multiplication methods resistant to power analysis. In *Cryptology ePrint Archive, Report 2005/222*, 2005. Available at: `http://eprint.iacr.org/2005/222`.
8. M. Feng, B.B. Zhu, C. Zhao, and S. Li. Signed MSB-set comb method for elliptic curve point multiplication. In K. Chen, R. Deng, X. Lai, and J. Zhou, editors, *Proceedings of Information Security Practice and Experience (ISPEC 2006)*, volume 3903 of *LNCS*, pages 13–24. Springer, 2006.
9. S. D. Galbraith, X. Lin, and M. Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. In *J. Cryptology*, volume 24(3), pages 446–469, 2011.
10. S.D. Galbraith, X. Lin, and M. Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. In A. Joux, editor, *Advances in Cryptology - EUROCRYPT*, volume 5479 of *LNCS*, pages 518–535. Springer, 2009.
11. R.P. Gallant, J.L. Lambert, and S.A. Vanstone. Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms. In J. Kilian, editor, *Advances in Cryptology - CRYPTO*, volume 2139 of *LNCS*, pages 190–200. Springer, 2001.
12. M. Hamburg. Fast and compact elliptic-curve cryptography. In *Cryptology ePrint Archive, Report 2012/309*, 2012. Available at: `http://eprint.iacr.org/2012/309`.
13. M. Hedabou, P. Pinel, and L. Beneteau. Countermeasures for preventing comb method against SCA attacks. In R. Deng, F. Bao, H. Pang, and J. Zhou, editors, *Proceedings of Information Security Practice and Experience (ISPEC 2005)*, volume 3439 of *LNCS*, pages 85–96. Springer, 2005.
14. H. Hisil, K. Wong, G. Carter, and E. Dawson. Twisted Edwards curves revisited. In J. Pieprzyk, editor, *Advances in Cryptology - ASIACRYPT*, volume 5350 of *LNCS*, pages 326–343. Springer, 2008.
15. P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In N. Koblitz, editor, *Advances in Cryptology - CRYPTO*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.
16. P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M. Wiener, editor, *Advances in Cryptology - CRYPTO*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.
17. C.H. Lim and P.J. Lee. More flexible exponentiation with precomputation. In Y. Desmedt, editor, *Advances in Cryptology - CRYPTO*, volume 839 of *LNCS*, pages 95–107. Springer, 1994.
18. ARM Limited. ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition, 2012.
19. P. Longa and C. Gebotys. Efficient techniques for high-speed elliptic curve cryptography. In S. Mangard and F.-X. Standaert, editors, *Proceedings of CHES 2010*, volume 6225 of *LNCS*, pages 80–94. Springer, 2010.
20. P. Longa and F. Sica. Four-dimensional Gallant-Lambert-Vanstone scalar multiplication. In X. Wang and K. Sako, editors, *Advances in Cryptology - ASIACRYPT*, volume 7658 of *LNCS*, pages 718–739. Springer, 2012.
21. P. Longa and F. Sica. Four-dimensional Gallant-Lambert-Vanstone scalar multiplication. In *Journal of Cryptology (to appear)*. Springer-Verlag, 2013.
22. B. Möller. Algorithms for multi-exponentiation. In S. Vaudenay and A.M. Youssef, editors, *Proceedings of SAC 2001*, volume 2259 of *LNCS*, pages 165–180. Springer, 2001.
23. T. Oliveira, D.F. López, J. Aranha, and F. Rodríguez-Henríquez. Two is the fastest prime. In *Cryptology ePrint Archive: Report 2013/131*, 2013. Available at: `http://eprint.iacr.org/2013/131`.
24. D.A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In D. Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006*, volume 3860, pages 1–20. Springer, 2006.
25. A.H. Sánchez and F. Rodríguez-Henríquez. NEON implementation of an attribute-based encryption scheme. In *Technical Report CACR 2013-07*, 2013. Available at: `http://cacr.uwaterloo.ca/techreports/2013/cacr2013-07.pdf`.
26. D. Weber and T.F. Denny. The solution of McCurley's discrete log challenge. In H. Krawczyk, editor, *Advances in Cryptology - CRYPTO*, volume 1462 of *LNCS*, pages 458–471. Springer, 1998.
27. T. Yanik, E. Savaş, and Ç.K. Koç. Incomplete reduction in modular arithmetic. In *IEE Proc. of Computers and Digital Techniques*, volume 149(2), pages 46–52, 2002.

A Formulas for Endomorphisms Φ and Ψ on Curve `Ted127-glv4`

Let $P = (X_1, Y_1, Z_1)$ be a point in homogeneous projective coordinates on a Twisted Edwards curve with eq. (1), $u = 1 + i$ be a quadratic non-residue in $\mathbb{F}_{p^2}$, and $\zeta_8 = u/\sqrt{-2a'}$ be a primitive 8th root of unity, where $a' = 27u^3(\frac{\sqrt{2}}{2} - 1)$. Then, we can compute $\Phi(P) = (X_2, Y_2, Z_2, T_2)$ as follows

$$X_2 = -X_1\left(\alpha Y_1^2 + \theta Z_1^2\right)\left[\mu Y_1^2 - \phi Z_1^2\right], \quad Y_2 = 2Y_1 Z_1^2\left[\phi Y_1^2 + \gamma Z_1^2\right],$$
$$Z_2 = 2Y_1 Z_1^2\left[\mu Y_1^2 - \phi Z_1^2\right], \quad T_2 = -X_1\left(\alpha Y_1^2 + \theta Z_1^2\right)\left[\phi Y_1^2 + \gamma Z_1^2\right],$$

where $\alpha = \zeta_8^3 + 2\zeta_8^2 + \zeta_8$, $\theta = \zeta_8^3 - 2\zeta_8^2 + \zeta_8$, $\mu = 2\zeta_8^3 + \zeta_8^2 - 1$, $\gamma = 2\zeta_8^3 - \zeta_8^2 + 1$ and $\phi = \zeta_8^2 - 1$.

For curve `Ted127-glv4`, we have the fixed values

$$\zeta_8 = 1 + Ai, \qquad \alpha = A + 2i, \qquad \theta = A + Bi,$$
$$\mu = (A-1) + (A+1)i, \qquad \gamma = (A+1) + (A-1)i, \qquad \phi = (B+1) + i,$$

where $A = 143485135153817520976780139629062568752$, $B = 170141183460469231731687303715884099729$.

Computing an endomorphism Φ with the formula above costs $12m + 2s + 5a$ or only $8m + 1s + 5a$ if $Z_1 = 1$. Similarly, we can compute $\Psi(P) = (X_2, Y_2, Z_2, T_2)$ as follows

$$X_2 = \zeta_8 X_1^p Y_1^p, \quad Y_2 = Z_1^{p^2}, \quad Z_2 = Y_1^p Z_1^p, \quad T_2 = \zeta_8 X_1^p Z_1^p.$$

Given the value for ζ_8 on curve `Ted127-glv4` computing an endomorphism Ψ with the formula above costs approximately $3m + 1s + 2M + 5A$ or only $1m + 2M + 4A$ if $Z_1 = 1$.

B Algorithms for Quadratic Extension Field Operations exploiting Interleaved ARM/NEON Operations

Below are the algorithms for multiplication and squaring over $\mathbb{F}_{p^2}$, with $p = 2^{127} - c$, for ARM platforms. They exploit functions interleaving ARM/NEON-based operations, namely `triple_mul_neonarm`, `double_mul_neonarm` and `double_red_neonarm`, detailed in Algorithms 7, 8 and 9, respectively.

Algorithm 12 Multiplication in $\mathbb{F}_{p^2}$ using completely or incompletely reduced inputs, $m = 3M_i + 9A_i + 2R$ (ARM platform).

Input: $a = (a_0 + a_1 i)$ and $b = (b_0 + b_1 i) \in \mathbb{F}_{p^2}$, where $0 \leq a_0, a_1, b_0, b_1 \leq 2^{127} - 1, p = 2^{127} - c, c$ small.
Output: $a \cdot b \in \mathbb{F}_{p^2}$.

1: $t_0 \leftarrow a_0 + a_1$	$[0, 2^{128} >$
2: $t_1 \leftarrow b_0 + b_1$	$[0, 2^{128} >$
3: $(T_0, T_1, T_2) \leftarrow$ `triple_mul_neonarm`$(a_0, b_0, a_1, b_1, t_0, t_1)$	$[0, 2^{256} >$
4: $C_0 \leftarrow T_0 - T_1$	$< -2^{254}, 2^{254} >$
5: **if** $C_0 < 0$, **then** $C_0 \leftarrow C_0 + 2^{128} \cdot p$	$[0, 2^{255} >$
6: $T_2 \leftarrow T_2 - T_0$	$[0, 2^{256} >$
7: $C_1 \leftarrow T_2 - T_1$	$[0, 2^{256} >$
8: **return** $(c_0, c_1) \leftarrow$ `double_red_neonarm`(C_0, C_1)	$[0, p >$

Algorithm 13 Squaring in $\mathbb{F}_{p^2}$ using completely reduced inputs, $s = 2M + 1A + 2A_i$ (ARM platform).

Input: $a = (a_0 + a_1 i) \in \mathbb{F}_{p^2}$, where $0 \leq a_0, a_1 \leq p - 1, p = 2^{127} - c, c$ small.
Output: $a^2 \in \mathbb{F}_{p^2}$.

1: $t_0 \leftarrow a_0 + a_1$	$[0, 2^{128} >$
2: $t_1 \leftarrow a_0 - a_1 \bmod p$	$[0, p >$
3: $t_2 \leftarrow a_0 + a_0$	$[0, 2^{128} >$
4: $(C_0, C_1) \leftarrow$ `double_mul_neonarm`(t_0, t_1, t_2, a_1)	$[0, p^2 >$
5: **return** $a^2 =$ `double_red_neonarm`(C_0, C_1)	$[0, p >$

C Cost of Fixed-Base Scalar Multiplication using the SAB-Set Comb Method

Below, we present estimated costs in terms of multiplications over $\mathbb{F}_{p^2}$ per bit for fixed-based scalar multiplication on curve `Ted127-glv4` using the SAB-set comb method (Algorithm 4). Precomputed points are stored as (x, y) coordinates ("affine") or as $(x + y, y - x, 2t)$ coordinates ("extended").

Table 3. Cost (in $\mathbb{F}_{p^2}$ multiplications per bit) of fixed-base scalar multiplication using the SAB-set comb method on curve `Ted127-glv4`.

v, w, # of points, memory	precomp coordinates	ARM Cortex-A9	ARM Cortex-A15	Intel Sandy Bridge	Intel Ivy Bridge
1, 6, 32 points, 2KB		2.88	3.24	3.05	3.08
2, 5, 32 points, 2KB	affine	**2.66**	**2.97**	**2.81**	**2.83**
4, 4, 32 points, 2KB		2.77	3.06	2.91	2.93
1, 6, 32 points, 3KB		2.76	3.11	2.92	2.94
2, 5, 32 points, 3KB	extended	**2.46**	2.73	**2.58**	**2.60**
4, 4, 32 points, 3KB		2.47	**2.71**	2.58	2.60
2, 6, 64 points, 4KB		2.33	2.63	2.46	2.49
4, 5, 64 points, 4KB	affine	**2.32**	**2.59**	**2.45**	**2.47**
8, 4, 64 points, 4KB		2.56	2.83	2.68	2.70
2, 6, 64 points, 6KB		2.21	2.50	2.33	2.35
4, 5, 64 points, 6KB	extended	**2.12**	**2.35**	**2.22**	**2.24**
8, 4, 64 points, 6KB		2.26	2.48	2.36	2.38
2, 7, 128 points, 8KB		2.26	2.60	2.40	2.43
4, 6, 128 points, 8KB	affine	**2.07**	**2.34**	**2.18**	**2.21**
8, 5, 128 points, 8KB		2.17	2.42	2.28	2.30
2, 7, 128 points, 12KB		2.25	2.60	2.37	2.40
4, 6, 128 points, 12KB	extended	**1.95**	2.20	**2.05**	**2.07**
8, 5, 128 points, 12KB		1.96	**2.17**	2.05	2.07
4, 7, 256 points, 16KB		2.02	2.34	2.14	2.18
8, 6, 256 points, 16KB	affine	**1.94**	**2.19**	**2.04**	**2.07**
16, 5, 256 points, 16KB		2.09	2.33	2.19	2.21
4, 7, 256 points, 24KB		2.02	2.34	2.12	2.15
8, 6, 256 points, 24KB	extended	**1.82**	**2.06**	**1.91**	**1.93**
16, 5, 256 points, 24KB		1.88	2.09	1.96	1.98
8, 7, 512 points, 32KB		1.92	2.22	2.03	2.06
16, 6, 512 points, 32KB	affine	**1.86**	**2.10**	**1.96**	**1.98**
32, 5, 512 points, 32KB		2.04	2.27	2.14	2.16
8, 7, 512 points, 48KB		1.91	2.22	2.00	2.04
16, 6, 512 points, 48KB	extended	**1.75**	**1.97**	**1.82**	**1.85**
32, 5, 512 points, 48KB		1.83	2.03	1.91	1.93

D Cost of Variable/Fixed-Base Double Scalar Multiplication on Curve `Ted127-glv4` using wNAF with Interleaving

Below, we present estimated costs in terms of multiplications over $\mathbb{F}_{p^2}$ per bit for variable/fixed-base double scalar multiplication on curve `Ted127-glv4` using wNAF with interleaving. Precomputations for the fixed base are stored as (x, y) coordinates ("affine") or as $(x+y, y-x, 2t)$ coordinates ("extended"). The window size $w_{2,j}$ for each sub-scalar j, number of points and memory listed in the first column correspond to requirements for the fixed base. For the variable base, we fix $w_1 = 4$, corresponding to the use of 16 precomputed points (see §3.3 and §6).

Table 4. Cost (in $\mathbb{F}_{p^2}$ multiplications per bit) of variable/fixed-base double scalar multiplication on curve `Ted127-glv4` using wNAF with interleaving, $w_1 = 4$, 16 "ephemeral" precomputed points.

$w_{2,j}$, # of points, memory	precomp coordinates	ARM Cortex-A9	ARM Cortex-A15	Intel Sandy Bridge	Intel Ivy Bridge
2/2/3/3, 6 points, 384 bytes	affine	6.66	7.28	7.11	7.14
2/2/2/2, 4 points, 384 bytes	extended	**6.57**	**7.14**	**7.00**	**7.03**
3/3/4/4, 12 points, 768 bytes	affine	6.07	6.64	6.51	6.53
3/3/3/3, 8 points, 768 bytes	extended	**5.95**	**6.47**	**6.36**	**6.38**
4/4/5/5, 24 points, 1.5KB	affine	5.71	6.24	6.13	6.15
4/4/4/4, 16 points, 1.5KB	extended	**5.58**	**6.07**	**5.98**	**6.00**
5/5/6/6, 48 points, 3KB	affine	5.42	5.92	5.82	5.84
5/5/5/5, 32 points, 3KB	extended	**5.33**	**5.80**	**5.72**	**5.74**
6/6/7/7, 96 points, 6KB	affine	5.20	5.68	5.59	5.61
6/6/6/6, 64 points, 6KB	extended	**5.08**	**5.53**	**5.46**	**5.48**
7/7/8/8, 192 points, 12KB	affine	5.05	5.52	5.44	5.46
7/7/7/7, 128 points, 12KB	extended	**4.95**	**5.40**	**5.33**	**5.35**
8/8/9/9, 384 points, 24KB	affine	4.98	5.44	5.37	5.39
8/8/8/8, 256 points, 24KB	extended	**4.83**	**5.26**	**5.20**	**5.22**