Noname manuscript No.
(will be inserted by the editor)

Efficient and Secure Algorithms for GLV-Based Scalar Multiplication and their Implementation on GLV-GLS Curves (Extended Version)

Armando Faz-Hernández · Patrick Longa · Ana H. Sánchez

Received: date / Accepted: date

Abstract We propose efficient algorithms and formulas that improve the performance of *side channel protected* elliptic curve computations with special focus on scalar multiplication exploiting the Gallant-Lambert-Vanstone (CRYPTO 2001) and Galbraith-Lin-Scott (EUROCRYPT 2009) methods. Firstly, by adapting Feng et al.'s recoding to the GLV setting, we derive new regular algorithms for variable-base scalar multiplication that offer protection against simple side-channel and timing attacks. Secondly, we propose an efficient, side-channel protected algorithm for fixed-base scalar multiplication which combines Feng et al.'s recoding with Lim-Lee's comb method. Thirdly, we propose an efficient technique that interleaves ARM and NEON-based multiprecision operations over an extension field to improve performance of GLS curves on modern ARM processors. Finally, we showcase the efficiency of the proposed techniques by implementing a state-of-the-art GLV-GLS curve in twisted Edwards form defined over $\mathbb{F}_{p^2}$, which supports a four dimensional decomposition of the scalar and is fully protected against timing attacks. Analysis and performance results are reported for modern x64 and ARM processors. For instance, we compute a variable-base scalar multiplication in 89,000 and 244,000 cycles on an Intel Ivy Bridge and an ARM Cortex-A15 processor (respect.); using a precomputed table of 6KB, we compute a fixed-base scalar multiplication in 49,000 and 116,000 cycles (respect.); and using a precomputed table of 3KB, we compute a double scalar multiplication in 115,000 and 285,000 cycles (respect.). The proposed techniques represent an important improvement of the state-of-the-art performance of elliptic curve computations, and allow us to set new speed records in several modern processors. The techniques also reduce the cost of adding protection against timing attacks in the computation of GLV-based variable-base scalar multiplication to below 10%.

This work is the extended version of a publication that appeared at CT-RSA 2014 [12].

Keywords Elliptic curves · scalar multiplication · side-channel protection · GLV method · GLS method · GLV-GLS curve · x64 processor · ARM processor · NEON vector unit.

Armando Faz-Hernández
Institute of Computing, University of Campinas, Brazil
E-mail: armfazh@ic.unicamp.br

Patrick Longa
Microsoft Research,
One Microsoft Way, Redmond, WA 98052, USA
E-mail: plonga@microsoft.com

Ana H. Sánchez
Digital Security Group,
Radboud University Nijmegen, The Netherlands
E-mail: ahsanchez@cs.ru.nl

1 Introduction

Let P be a point of prime order r on an elliptic curve over $\mathbb{F}_p$ containing a degree-2 endomorphism ϕ. The Gallant-Lambert-Vanstone (GLV) method computes the scalar multiplication kP as $k_1P + k_2\phi(P)$ [17]. If k_1, k_2 have approximately half the bitlength of the original scalar k, one should expect an elimination of half the number of doublings by using the Straus-Shamir simultaneous multi-scalar multiplication technique. Thus, the method is especially useful for speeding up the case in which the base point P is variable, known as variable-base scalar multiplication. Later, in [16] Galbraith et al. showed how to exploit the Frobenius endomorphism to enable the use of the GLV approach on a wider set of curves defined over the quadratic extension field $\mathbb{F}_{p^2}$. Since then, significant research has been performed to improve the performance [30,24] and to explore the applicability to other settings [20,35] or to higher dimensions on genus one curves [24,31,18] and genus two curves [8,9,18]. Unfortunately, most of the work and comparisons with other approaches have been carried out with *unprotected* algorithms and implementations. In fact, little effort has been done to investigate methods for protecting GLV-based implementations against side-channel attacks. Just recently, Longa and Sica [31] used the regular windowed recoding by Okeya and Takagi [34] in combination with interleaving [17,33] to make a four-dimensional implementation constant time. However, the use of this standard approach in the GLV paradigm incurs a high cost in terms of storage and computing performance because of the high number of required precomputations. This issue worsens for higher dimensions [9].

In this work, we propose a new signed representation, called GLV-based Sign-Aligned Column (GLV-SAC), that gives rise to a new method for scalar multiplication using the GLV method. We depart from the traditional approach based on interleaving or joint sparse form and adapt the recoding by Feng et al. [13], which was originally intended for standard comb-based fixed-base scalar multiplication, to the computation of GLV-based variable-base scalar multiplication. The method supports a regular execution as required to protect implementations against some simple side-channel attacks such as simple power analysis (SPA) [27]. Moreover, it does not require dummy operations, making it resilient to safe-error attacks [42,43], and can be used as basis for realizing constant-time implementations that guard against timing attacks [26,11,2,36]. In addition, we present different variants of the technique that are intended for different scenarios exploiting simple or complex GLV decompositions, and thus provide algorithms that have broad applicability to many settings using GLV, GLS, or a combination of both [16,20,30, 24,31,35,8,9,18,39]. In comparison with the best previous approaches, the method improves the computing performance especially during the potentially expensive precomputation stage, and allows us to save *at least* half of the storage requirement for precomputed values without impacting performance. For instance, the method injects a 17% speedup in the overall computation and a 78% reduction in the memory consumption for a GLV-GLS curve using a 4-GLV decomposition (see §6). The savings in memory without impacting performance are especially relevant for the deployment of GLV-based implementations in constrained devices. Depending on the cost of endomorphisms, the improvement provided by the method is expected to increase for higher-degree decompositions.

Besides variable-base scalar multiplication, there are two other core computations that are the basis of most curve-based protocols: kP with P known in advance (fixed-base multiplication), and $kP+lQ$ with P known in advance and Q unknown (fixed/variable-base double scalar multiplication), where P and Q are points belonging to an elliptic curve (sub)group of prime order r and k, l are integers in $[1, r-1]$. For example, fixed-base scalar multiplication is used during signature generation and fixed/variable-base double scalar multiplication is used during signature verification in ECDSA.

For the case of the fixed-base scenario, we introduce an optimized variant of the side-channel protected fixed-base comb method by Feng et al. [13] that exploits the original, multi-table Lim-Lee's comb technique [28]. Our approach is similar in spirit to Hamburg's comb technique [19] which uses Hedabou et al.'s representation [22]. Our algorithm is generic (i.e., it does not exploit endomorphisms) and is as efficient as Hamburg's algorithm during the on-line computation but has a slightly cheaper off-line precomputation phase. The algorithm has already been exploited in the implementation of the new elliptic curves recently proposed by Bos et al. in [10] and integrated to the MSR Elliptic Curve Cryptography Library (MSR ECCLib) [37].

Processors based on the ARM architecture are widely used in modern smartphones and tablets due to their low power consumption. 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 certain applications, some ARM processors include a powerful set of vector instructions known as NEON. This consists of a 128-bit Single Instruction Multiple Data (SIMD) engine that includes 16 128-bit registers. Recent research has exploited NEON to accelerate cryptographic operations [7,19,38]. On one hand, the interleaving of ARM and NEON instructions is a well-known technique (with increasing potential on modern processors) that can be exploited in cryptography; e.g., see [7]. On the other hand, vectorized implementations using NEON can perform efficiently the computation of independent multiplications (as found in operations over $\mathbb{F}_{p^2}$); e.g., see [38]. In this work, we take these optimizations further and propose a technique that interleaves ARM- and NEON-based multiprecision operations, such as multiplication, squaring and modular reduction, in extension field operations in order to maximize the inherent parallelism and hide the execution latency. The technique is especially relevant for implementing the quadratic extension field layer in GLS curves [16] and pairing computations [1]. For instance, it injects a significant speedup in the range 17%-34% in the scalar multiplication execution on a GLV-GLS curve (see §5 and §6).

To demonstrate the efficiency of our techniques, we implement the state-of-the-art twisted Edwards GLV-GLS curve over $\mathbb{F}_{p^2}$ with $p = 2^{127} - 5997$ recently proposed by Longa and Sica [31]. This curve, referred to as `Ted127-glv4`, supports a 4-GLV decomposition. Moreover, we also present efficient algorithms for implementing field and quadratic extension field operations targeting our 127-bit prime on x64 and ARM platforms. We combine incomplete reduction [41] and lazy reduction [40], expanding techniques by [30]. These optimized operations are then applied to state-of-the-art twisted Edwards formulas [3,23] to speed up computations in the setting of curves over $\mathbb{F}_{p^2}$. Our implementations of the *three* core scalar multiplication scenarios, namely, variable-base, fixed-base and double scalar, target modern x64 and ARM processors and include full protection against timing attacks.

We show that the proposed algorithms and formulas reduce significantly the cost of adding protection against timing attacks and the storage requirement for precomputations, and allow us to set a new speed record for protected software. For instance, a protected variable-base elliptic curve scalar multiplication on curve `Ted127-glv4` runs in 96,000 cycles on an Intel Sandy Bridge machine (Windows OS), using only 1KB of memory for precomputed values. This is 30% faster, using almost 1/5 of the storage, than the state-of-the-art implementation reported by Longa and Sica [31] that computes the same operation in 137,000 cycles using 4.5KB of memory for precomputations. Moreover, this result is only 5% slower, using 1/2 of the storage, than the state-of-the-art *unprotected* computation in [31], which runs in 91,000 cycles using 2KB of memory. The performance of the variable-base computation is even faster on Linux: the operation runs in 92,000 cycles on the same Sandy Bridge machine. These results not only represent a new speed record for protected software but also mark the first time that a constant-time 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 exploiting the technique that interleaves NEON and ARM-based operations (see §6 for full benchmark results).

This paper is organized as follows. In §2, we give some preliminaries about the GLV and GLS methods, side-channel attacks and the protected comb methods by Feng et al. [13,14] and Hedabou et al. [22]. In §3, we present the new GLV-based representation, its variants and the corresponding scalar multiplication method. In §4, we describe the new algorithm for fixed-base scalar multiplication. We describe the implementation of curve `Ted127-glv4` as well as optimized algorithms for field, extension field and point operations targeting x64 and ARM platforms in §5. In this section, we also discuss the interleaving technique for ARM. Finally, in §6, we perform an analysis of the proposed methods and present benchmark results of the core scalar multiplication scenarios on several x64 and ARM processors.

2 Preliminaries

2.1 The GLV and GLS Methods

In this section, we briefly describe the GLV and GLS methods in a generic, m dimensional framework. Let C be a curve defined over a finite field $\mathbb{F}_p$ equipped with an efficiently computable endomorphism ϕ. The GLV method to compute scalar multiplication [17] consists of first decomposing the scalar k into sub-scalars k_i for $0 \leq i < m$ and then computing $\sum_{i=0}^{m-1} k_i D_i$ using the Straus-Shamir trick for simultaneous multi-scalar multiplication, where D_0 is the input divisor from the divisor class group of the curve and $D_i = \phi^i(D_0)$. If all of the sub-scalars have approximately the same bitlength, the number of required doublings is reduced to approximately $\log_2 r/m$, where r is the prime order of the curve subgroup. Special curves equipped with endomorphisms which are different to the Frobenius endomorphism are known as GLV curves.

The GLS method [16,15] lifts the restriction to special curves and exploits an endomorphism ψ arising from the p-power Frobenius endomorphism on a wider set of curves C' defined over an extension field $\mathbb{F}_{p^k}$ that are $\mathbb{F}_{p^n}$-isogenous to curves $C/\mathbb{F}_p$, where $k|n$. Equipped with ψ to perform the scalar decomposition, one then proceeds to apply the GLV method as above. More complex decompositions arise by applying the GLS paradigm to GLV curves (a.k.a. GLV-GLS curves [16,31]).

These techniques have received lots of attention recently, given their significant impact in the performance of curve-based systems. Longa and Gebotys [30] report efficient implementations of GLS curves over $\mathbb{F}_{p^2}$ using 2 dimensional decompositions. In [24], Hu, Longa and Xu explore a GLV-GLS curve over $\mathbb{F}_{p^2}$ supporting a 4 dimensional decomposition. In [8], Bos et al. study 2 and 4 dimensional decompositions on genus 2 curves over $\mathbb{F}_p$. Bos et al. [9] explore the combined GLV-GLS approach over genus 2 curves defined over $\mathbb{F}_{p^2}$, which supports an 8-GLV decomposition. In the case of binary GLS elliptic curves, Oliveira et al. [35] report the implementation of a curve exploiting the 2-GLV method. More recently, Guillevic and Ionica [18] show how to exploit the 4-GLV method on certain genus one curves defined over $\mathbb{F}_{p^2}$ and genus two curves defined over $\mathbb{F}_p$; and Smith [39] proposes a new family of elliptic curves that support 2-GLV decompositions.

From all the works above, only [31] and [35] include side-channel protection in their GLV-based implementations.

2.2 Side-Channel Attacks and Countermeasures

Side-channel attacks [26] exploit leakage information obtained from the physical implementation of a cryptosystem to get access to private key material. Examples of physical information that can be exploited are power, time, electromagnetic emanations, among others. In particular, much attention has been put on timing [26,11] and simple power attacks (SPA) [27], given their broad applicability and relatively low costs to be realized in practice. Traditionally, the different attacks can also be distinguished by the number of traces that are exploited in the analysis: simple side-channel attacks (SSCA) require only one trace (or very few traces) to observe the leakage that directly reveals the secret bits, whereas differential side-channel attacks (DSCA) require many traces to perform a statistical analysis on the data. The feasibility of these attacks depends on the targeted application, but it is clear that SSCA attacks are feasible in a wider range of scenarios. In this work, we focus on methods that minimize the risk posed by timing attacks and SSCA attacks such as SPA.

In curve-based cryptosystems, the first step to achieve protection against these attacks is to use regular algorithms for performing scalar multiplication (other methods involve the use of unified formulas, but these are generally expensive). One efficient approach in this direction is to recode the scalar to a representation exhibiting a regular pattern. In particular, for the case of variable-base scalar multiplication, the regular windowed recoding proposed by Okeya and Takagi [34] and further analyzed by Joye and Tunstall [25] represents one of the most efficient alternatives. Nevertheless, in comparison with the standard width-w non-adjacent form (wNAF) [21] used in unprotected implementations, the Okeya-Takagi recoding increases the nonzero density from $1/(w+1)$ to $1/(w-1)$. In contrast, side-channel protected methods for scalar multiplication exploiting the GLV method have not been fully studied. Furthermore, we note that methods typically efficient in the standard case are not necessarily efficient in the GLV paradigm. For example, in [31], Longa and Sica apply the Okeya-Takagi recoding to protect scalar multiplication on a GLV-GLS curve using a 4 dimensional GLV decomposition against timing attacks. The resulting protected implementation is about 30% more expensive than the unprotected version. In this work, we aim at reducing that gap, providing efficient methods that can be exploited to improve and protect GLV and GLS-based implementations.

The comb method [28] is an efficient approach for the case of fixed-base scalar multiplication. However, in its original form, the method is unprotected against SSCA and timing attacks. An efficient approach to achieve a regular execution is to recode the scalar using signed nonzero representations such as LSB-set [13], MSB-set [14] or SAB-set [22]. A key observation in this work is that the basic version of the fixed-base comb execution (i.e., without exploiting multiple tables) has several similarities with a GLV-based variable-base execution. So it is therefore natural to adapt these techniques to the GLV setting to achieve side-channel protection. In particular, the LSB-set representation is a good candidate, given that an analogue of this method in the GLV setting minimizes the cost of precomputation.

2.3 The Least Significant Bit - Set (LSB-Set) Representation and Variants

Feng, Zhu, Xu and Li [13] proposed a clever signed representation, called LSB-set, that is based on the equivalence $1 \equiv 1\bar{1}\ldots\bar{1}$ (assuming the notation $-1 \equiv \bar{1}$). They used this representation to protect the comb method [28] in the computation of fixed-base scalar multiplication (we refer to this method as LSB-set comb scalar multiplication). Next, we briefly describe the LSB-set recoding and its application to fixed-base scalar multiplication. The reader is referred to [28] and [13] for complete details about the original comb method and the LSB-set comb method, respectively.

Let t be the bitlength of the prime order r of a given curve subgroup, such that possible scalars k in the computation of scalar multiplication are in the range $[0, r-1]$. Assume that a given scalar k is partitioned in w consecutive parts of $d = \lceil t/w \rceil$ bits each (padding k with $(dw - t)$ zeros to the left as necessary). Let the updated binary representation of k be $(k_{l-1}, k_{l-2}, \ldots, k_0)$, where $l = dw$. One can visualize the bits of k in matrix form by considering the w pieces as the rows with the least significant part on top and the most significant part at the bottom. The LSB-set recoding consists of first replacing every sequence $00\ldots01$ in the top row by $1\bar{1}\ldots\bar{1}\bar{1}$ (keeping the same number of digits). Then, it involves converting every bit k_i in the remaining rows in such a way that output digits b_i for $d \le i \le (l-1)$ are in the digit set $\{0, b_{i \bmod d}\}$. That is, digits in the same column of the recoded matrix are either 0 or share the same sign. After precomputing all the possible multiples of the base point corresponding to a "digit-column", one can proceed to compute a comb fixed-base scalar multiplication by scanning the digit-columns of the recoded matrix from left to right. Since every digit-column is nonzero by definition, the computation consists of a point doubling and an addition with a precomputed value at every iteration, providing a regular execution that is protected against simple side-channel attacks.

There are other variants in the literature that have also been exploited for implementing protected comb fixed-base scalar multiplication. Feng, Zhu, Zhao and Li proposed in [14] the Most Significant Bit - Set (MSB-set) representation, which reduces slightly the cost in comparison with the LSB-set comb method for the case in which $w \mid t$. The main difference with LSB-set resides in that MSB-set applies the transformation $1 \mapsto 1\bar{1}\ldots\bar{1}$ to the most significant d bits of the scalar, instead of the least significant portion. In [22], Hedabou, Pinel and Beneteau proposed a *full* signed nonzero representation, referred to as Signed All-Bit-Set (SAB-set), that uses the above transformation to recode the whole scalar. In this case, the cost of precomputation is expected to be slightly higher since no zeros are used in the representation. We comment that the algorithms presented in this work can be modified to use the MSB-set or SAB-set representations.

3 The GLV-Based Sign-Aligned Column (GLV-SAC) Representation

In this section, we introduce a variant of the LSB-set recoding that is amenable for the computation of side-channel protected variable-base scalar multiplication in the GLV setting. The new recoding is called GLV-Based Sign-Aligned Column (GLV-SAC). Also, we present a new method for GLV-based scalar multiplication exploiting the proposed representation.

In the following, we first discuss the GLV-SAC representation in a generic setting. In Section 3.2, we discuss variants that are expected to be more efficient when $m = 2$ and $m \ge 8$. To simplify the descriptions, we assume in the remainder that we are working on an elliptic curve. The techniques and algorithms can be easily extended to other settings such as genus 2 curves.

Let $k_s = \{k_0, k_1, \ldots, k_j, \ldots, k_{m-1}\}$ be a set of positive sub-scalars in the setting of GLV with dimension m. The basic idea of the new recoding is to have one of the sub-scalars of the m-GLV decomposition, say $k_J \subset k_s$, represented in signed nonzero form and acting as a "sign-aligner". The latter means that k_J determines the sign of all the digits of remaining sub-scalars according to their relative position.

The GLV-SAC representation has the following properties:

(i) The length of the digit representation of every sub-scalar $k_j \in k_s$ is fixed and given by $l = \lceil \log_2 r/m \rceil + 1$, where r is the prime subgroup order.
(ii) Exactly one sub-scalar, which should be odd, is expressed by a signed nonzero representation $k_J = (b^J_{l-1}, \ldots, b^J_0)$, where all digits $b^J_i \in \{1, -1\}$ for $0 \le i < l$.
(iii) Sub-scalars $k_j \in k_s \setminus \{k_J\}$ are expressed by signed representations $(b^j_{l-1}, \ldots, b^j_0)$ such that $b^j_i \in \{0, b^J_i\}$ for $0 \le i < l$.

In the targeted setting, (i) and (ii) guarantee a constant-time execution regardless of the value of the scalar k and without having to appeal to masking for dealing with the identity element. Item (iii) allows us to reduce the size of the precomputed table by a factor of 2, while minimizing the cost of precomputation.

Note that we do not impose any restriction on which sub-scalar should be designated as k_J. In some settings, choosing any of the k_j (with the exception of the one corresponding to the base point, i.e., k_0) could lead to the same performance in the precomputation phase and be slightly faster than $k_J = k_0$, if one takes into consideration the use of mixed point additions. The condition that k_J should be odd enables the conversion of any integer to a full signed nonzero representation using the equivalence $1 \equiv 1\bar{1}\ldots\bar{1}$. To deal with this restriction during the scalar multiplication, we first convert the selected sub-scalar k_J to odd (if even), and then make the corresponding correction in the end (refer to Section 3.1 for more details). Finally, the reader should note that the GLV-SAC representation, in the way we describe it above, assumes that the sub-scalars are all positive. This restriction is imposed in order to achieve the minimum length $l = \lceil \log_2 r/m \rceil + 1$ in the representation. We lift this restriction in Section 3.3.

An efficient algorithm to recode the sub-scalars to GLV-SAC proceeds as follows. Assume that each sub-scalar k_j is padded with zeros to the left until reaching the fixed length $l = \lceil \log_2 r/m \rceil + 1$. After choosing a suitable k_J to act as the "sign-aligner", the sub-scalar k_J is recoded to signed nonzero digits b_i^J using the equivalence $1 \equiv 1\bar{1}\ldots\bar{1}$, i.e., every sequence $00\ldots01$ is replaced by a sequence $1\bar{1}\ldots\bar{1}\bar{1}$ with the same number of digits. Remaining sub-scalars are then recoded in such a way that output digits at position i are in the set $\{0, b_i^J\}$, i.e., nonzero digits at the same relative position share the same sign. This is shown as Algorithm 1.

We highlight that, in contrast to [13, Alg. 4] and [14, Alg. 2], our recoding algorithm is simpler and exhibits a regular and constant-time execution, making it resilient to timing attacks. Moreover, Algorithm 1 can be implemented very efficiently by exploiting the fact that the only purpose of the recoded digits from the sub-scalar k_J is, by definition, to determine the sign of their corresponding digit-columns (see details in Alg. 2). Since $k_{i+1}^J = 0$ and $k_{i+1}^J = 1$ indicate that the corresponding output digit-column i will be negative and positive, respectively, Step 3 of Algorithm 1 can be reduced to $b_i^J = k_{i+1}^J$ by assuming the convention $b_i^J = 0$ to indicate negative and $b_i^J = 1$ to indicate positive, for $0 \le i < l$. Following this convention, further efficient simplifications are possible for Steps 6 and 7.

Algorithm 1 Protected Recoding Algorithm for the GLV-SAC Representation.

Input: m l-bit positive integers $k_j = (k_{l-1}^j, \ldots, k_0^j)_2$ for $0 \le j < m$, an odd "sign-aligner" $k_J \in \{k_j\}^m$, where $l = \lceil \log_2 r/m \rceil + 1$, m is the GLV dimension and r is the prime subgroup order.
Output: $(b_{l-1}^j, \ldots, b_0^j)_{\text{GLV-SAC}}$ for $0 \le j < m$, where $b_i^J \in \{1, -1\}$, and $b_i^j \in \{0, b_i^J\}$ for $0 \le j < m$ with $j \ne J$.

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

3.1 GLV-Based Scalar Multiplication using GLV-SAC

We now present a new method for computing GLV-based variable-base scalar multiplication using the GLV-SAC representation (see Algorithm 2). To simplify the description, we assume that k_0 is fixed as the "sign-aligner" k_J (it is easy to modify the algorithm to set any other sub-scalar to k_J). The basic idea is to arrange the sub-scalars, after being converted to their GLV-SAC representation, in matrix form from top to bottom, with sub-scalar $k_J = k_0$ at the top, and then run a simultaneous multi-scalar multiplication execution scanning digit-columns from left to right. When using the GLV-SAC recoding, every digit-column i is expected to be nonzero and has 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 $1 \le j < m$ and $0 \le i < l$. Since nonzero digits in the same column have the same sign, one only needs to precompute all the positive combinations $P_0 + u_1P_1 + \ldots + u_{m-1}P_{m-1}$ with $u_j \in \{0, 1\}$, where P_j are the base points of the sub-scalars. Assuming that negation of group elements is inexpensive in a given curve subgroup, negative values can be computed on-the-fly during the evaluation stage.

Algorithm 2 Protected m-GLV Variable-Base Scalar Multiplication using the GLV-SAC 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 $0 \leq j < m$, $l = \lceil \frac{\log_2 r}{m} \rceil + 1$ and $\max(t_j) = \lceil \frac{\log_2 r}{m} \rceil$.
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: Set $k_J = k_0$. Pad each $k_j$ with $(l - t_j)$ zeros to the left for $0 \leq j < m$ and convert them to the GLV-SAC representation using Algorithm 1 such that $k_j = (b^j_{l-1}, \ldots, b^j_0)_{\text{GLV-SAC}}$. 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:     $Q = 2Q$
8:     $Q = Q + s_i P[\mathbb{K}_i]$
9: if even $= 0$ then $Q = Q + P_0$
10: return $Q$
```

Since the GLV-SAC recoding requires that the "sign-aligner" k_J (in this case, k_0) be odd, k_0 is subtracted by one if it is even in Step 3 of Algorithm 2. The correction is then performed at the end of the evaluation stage at Step 10. 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, in the implementation discussed in Section 6, the value $P[\mathbb{K}_i]$ required at Step 9 is retrieved from memory by performing a linear pass over the whole precomputed table using conditional move instructions. The final value $s_i P[\mathbb{K}_i]$ is then obtained by performing a second linear pass over the points $P[\mathbb{K}_i]$ and $-P[\mathbb{K}_i]$. Similarly, to realize Step 10, we always carry out the computation $Q' = Q + P_0$ and then 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 other settings, one requires to calculate the sub-scalars in a decomposition procedure. We remark that this computation should also be performed in constant time for protecting against timing attacks (e.g., see the details for `Ted127-glv4` in §6).

Example 1. Let $m = 4, \log_2 r = 16$ and $kP = 11P_0 + 6P_1 + 14P_2 + 3P_3$. Using Algorithm 1, the corresponding GLV-SAC representation of fixed length $l = \lceil 16/4 \rceil + 1 = 5$ is given by (arranged in matrix form from top to bottom as required in Algorithm 2)

$$\begin{bmatrix} k_0 \\ k_1 \\ k_2 \\ k_3 \end{bmatrix} \equiv \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \equiv \begin{bmatrix} 1 & \bar{1} & 1 & \bar{1} & 1 \\ 1 & \bar{1} & 0 & \bar{1} & 0 \\ 1 & 0 & 0 & \bar{1} & 0 \\ 0 & 0 & 1 & \bar{1} & 1 \end{bmatrix}$$

According to Algorithm 2, digit columns are given by $\mathbb{K}_0 = [100] = 4, \mathbb{K}_1 = [\bar{1}\bar{1}\bar{1}] = 7, \mathbb{K}_2 = [100] = 4, \mathbb{K}_3 = [00\bar{1}] = 1$ and $\mathbb{K}_4 = [011] = 3$, and their corresponding s_i are $s_0 = 1, s_1 = -1, s_2 = 1, s_3 = -1$ and $s_4 = 1$. Precomputed values $P[u]$ are given by $P[0] = P_0, P[1] = P_0 + P_1, P[2] = P_0 + P_2, P[3] = P_0 + P_1 + P_2, P[4] = P_0 + P_3, P[5] = P_0 + P_1 + P_3, P[6] = P_0 + P_2 + P_3$ and $P[7] = P_0 + P_1 + P_2 + P_3$. At Step 5 of Alg. 2, we set $Q = s_4 P[\mathbb{K}_4] = P[3] = P_0 + P_1 + P_2$. The main loop in the evaluation stage is then executed as shown in Table 1.

Cost Analysis. In order to simplify comparisons, let us consider here a setting in which precomputed points are left in some projective system. When converting points to affine is convenient, one should include the cost of this conversion. Also, the analysis below does not consider optimizations exploiting cheap endomorphism mappings during precomputation, since this is dependent on a specific application. The reader is referred to Section 6 for a more precise comparison in a practical implementation using a twisted Edwards GLV-GLS curve.

The cost of the proposed m-GLV variable-base scalar multiplication using the GLV-SAC representation (Alg. 2) is given by $(l-1)$ doublings and l additions during the evaluation stage using 2^{m-1} points, where $l = \lceil \frac{\log_2 r}{m} \rceil + 1$. Naively, precomputation costs $2^{m-1} - 1$ additions (in practice, several of these additions might be performed using cheaper mixed additions). So the total cost is given by $(l-1)$ doublings and $(l + 2^{m-1} - 1)$ additions.

In contrast, the method based on the regular windowed recoding [34] used in [31] requires $(l-1)$ 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 during the precomputation stage, using $m \cdot (2^{w-2} + 1)$ points (naive approach

i	3	2	1	0
$2Q$	$2P_0 + 2P_1 + 2P_2$	$2P_0 + 2P_1 + 4P_2$	$6P_0 + 4P_1 + 8P_2 + 2P_3$	$10P_0 + 6P_1 + 14P_2 + 2P_3$
$Q + s_iP[\mathbb{K}_i]$	$P_0 + P_1 + 2P_2$	$3P_0 + 2P_1 + 4P_2 + P_3$	$5P_0 + 3P_1 + 7P_2 + P_3$	$11P_0 + 6P_1 + 14P_2 + 3P_3$

Table 1 Execution of the main loop of Algorithm 2 in Example 1.

without exploiting endomorphisms). If, for example, $r = 256, m = 4$ and $w = 5$ (typical parameters to achieve 128-bit security on a curve similar to `Ted127-glv4`), the new method costs 64 doublings and 72 additions using 8 points, whereas the regular windowed method costs 68 doublings and 99 additions using 36 points. Thus, the new method improves performance while reduces dramatically the number of precomputations (in this case, to almost 1/5 of the storage). Assuming that one addition costs 1.3 doublings, the expected speedup is 20%.

Certainly, one can reduce the number of precomputations when using the regular windowed recoding by only precomputing multiples corresponding to one or some of the sub-scalars. However, these savings in memory come at the expense of computing endomorphisms during the evaluation stage, which can cost from several multiplications [8] to approximately one full point addition each (see Appendix B). The proposed method always requires the minimum storage without impacting performance.

The basic GLV-SAC representation and its corresponding scalar multiplication are particularly efficient for 4-dimensional GLV. In the following section, we discuss variants that are efficient for $m = 2$ and $m \geq 8$.

3.2 Windowed and Partitioned GLV-SAC: Case of Dimension 2 and 8 (or Higher)

In some cases, the performance of the proposed scalar multiplication can be improved further by combining windowed techniques with the GLV-SAC recoding. Given a window width w, assume that every sub-scalar in a set $k_s = \{k_j\}^m$ has been padded with enough zeros to the left to guarantee that $w|l$, where $l = (\lceil \log_2 r/w \rceil + 1) \bmod w + (\lceil \log_2 r/w \rceil + 1)$ is the expected length of every recoded sub-scalar using an extended GLV-SAC representation that we refer to as wGLV-SAC. The basic idea is to join every w consecutive digits after applying the GLV-SAC recoding, and precompute all possible values $P[u] = u'P_0 + u_0P_1 + \ldots + u_{m-2}P_{m-1}$ for each $u' \in \{1, 3, \ldots, 2^w - 1\}$ (i.e., $0 \leq u < 2^{wm-1}$). Note that possible values for $u_0, \ldots, u_{m-2}$ should be fixed according to the restriction that all the digits in the same "column" share the same sign. For example, let us assume that $m = w = 2$. Then, possible two-digit values u' for k_0 are $(1\bar{1}) \equiv 1$ and $(11) \equiv 3$. For $u' = 1$, possible two-digit values u_0 for k_1 are $(0\bar{1}) \equiv -1$, $(00) \equiv 0$, $(1\bar{1}) \equiv 1$ and $(10) \equiv 2$. These values correspond to table entries $u'P_0 + u_0P_1$: $P[0] = P_0 - P_1$, $P[1] = P_0$, $P[2] = P_0 + P_1, P[3] = P_0 + 2P_1$. Similarly, for $u' = 3$, possible two-digit values u_0 for k_1 are $(00) \equiv 0$, $(01) \equiv 1$, $(10) \equiv 2$ and $(11) \equiv 3$. These values correspond to table entries $u'P_0 + u_0P_1$: $P[4] = 3P_0$, $P[5] = 3P_0 + P_1$, $P[6] = 3P_0 + 2P_1, P[7] = 3P_0 + 3P_1$. Again, points corresponding to negative values of u' do not need to be stored in the table because they can be computed on-the-fly. Conveniently, Algorithm 1 can also be used to obtain wGLV-SAC(k_j), with the only change in the fixed length to $l = (\lceil \log_2 r/w \rceil + 1) \bmod w + (\lceil \log_2 r/w \rceil + 1)$. After conversion to the wGLV-SAC representation, scalar multiplication then proceeds by scanning w-digit columns in the recoded matrix from left to right.

Example 2. Let $m = 2, \log_2 r = 8$, $w = 2$ and $kP = 11P_0 + 14P_1$. Using Algorithm 1, the corresponding wGLV-SAC representation with fixed length $l = (\lceil 8/2 \rceil + 1) \bmod 2 + \lceil 8/2 \rceil + 1 = 6$, arranged in matrix form from top to bottom, is given by

$$\begin{bmatrix} k_0 \\ k_1 \end{bmatrix} \equiv \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \equiv \left[\begin{array}{cc|cc|cc} 1 & \bar{1} & \bar{1} & 1 & \bar{1} & 1 \\ 1 & \bar{1} & 0 & 0 & \bar{1} & 0 \end{array}\right]$$

In this case, one can consider the following convention for the 2-digit columns: $\mathbb{K}_i = [b^1_{2i+1}b^1_{2i}, b^0_{2i+1}b^0_{2i}] \equiv |2b^1_{2i+1} + b^1_{2i}| + (3 \cdot |2b^0_{2i+1} + b^0_{2i}| + 1)/2 - 1$ with signs $s_i = b^0_{2i+1}$, for $0 \leq i < \frac{l}{2}$ (using the notation $k_j = (b^j_{l-1}, \ldots, b^j_0)_{w\text{GLV-SAC}}$). Thus, the 2-digit columns are given by $\mathbb{K}_0 = [\bar{2}, \bar{1}] \equiv 3$, $\mathbb{K}_1 = [0, \bar{1}] \equiv 1$ and $\mathbb{K}_2 = [1, 1] \equiv 2$, and their corresponding s_i are $s_0 = -1$, $s_1 = -1$ and $s_2 = 1$. As explained in the text before the example, precomputed values are given by $P[u] = u'P_0 + u_0P_1$ for $0 \leq u < 8$ and $u' \in \{1, 3\}$. Explicitly, $P[0] = P_0 - P_1$, $P[1] = P_0$, $P[2] = P_0 + P_1, P[3] = P_0 + 2P_1, P[4] = 3P_0, P[5] = 3P_0 + P_1, P[6] = 3P_0 + 2P_1$ and $P[7] = 3P_0 + 3P_1$. In the evaluation stage we first set $Q = s_2P[\mathbb{K}_2] = P[2] = P_0 + P_1$ and then execute:

i	1	0
2^wQ	$4P_0 + 4P_1$	$12P_0 + 16P_1$
$Q + s_iP[\mathbb{K}_i]$	$3P_0 + 4P_1$	$11P_0 + 14P_1$

Since the requirement of precomputations, given by 2^{wm-1}, increases rapidly as w and m grow, windowed GLV-SAC is especially attractive for 2-dimensional GLV implementations. In this case, by fixing $w = 2$ the number of precomputed points is only 8. At the same performance level (in terms of number of additions and doublings in the evaluation stage), this is approximately half the memory requirement of a method based on the regular windowed recoding [34] [1].

Whereas joining columns in the representation matrix is amenable for small m using windowing, for large m it is recommended to join rows instead. We illustrate the approach with $m = 8$. Given a set of sub-scalars $k_s = \{k_j\}^m$ for $0 \leq j < 8$, we first partition it in c consecutive sub-sets k'_i such that $c|8$, and then convert every sub-set to the GLV-SAC representation (using Algorithm 1). In this case, every column in the matrix consists of c sub-columns, each one corresponding to a sub-set k'_i. Scalar multiplication then proceeds by scanning c sub-columns per iteration from left to right. Thus, with this "partitioned" GLV-SAC approach, one increases the number of point additions per iteration in the main loop of Algorithm 2 from one to c. However, the number of required precomputations is reduced significantly from 2^{m-1} to $c \cdot 2^{\frac{m}{c}-1}$. For example, for $m = 8$, this introduces a reduction in the number of points from 128 to only 16 if c is fixed to 2 (each sub-table corresponding to a sub-set of scalars contains 8 points). At the same performance level (in terms of number of additions and doublings in the evaluation stage), this is approximately half the memory requirement of a method based on the regular windowed recoding [34], as discussed by the recent work by Bos et al. [9]. Performance is also expected to be improved since the number of point operations in the precomputation stage is significantly reduced. Note that, if one only considers positive sub-scalars and the endomorphism mapping is inexpensive in comparison to point addition, then sub-tables can be computed by simply applying the endomorphism to the first sub-table arising from the base point P_0. In some instances, such as the 8-GLV in [9], this approach is expected to reduce further the cost of precomputation. However, an issue arises when sub-scalars can also be negative. We solve this problem in the next subsection.

[1] However, in some cases one can afford the reduction of precomputations from 16 to 8 when using the windowed recoding if endomorphisms are cheap and can be computed on-the-fly during the evaluation stage; e.g., see [35].

3.3 Extended GLV-SAC Representation: Dealing with Negative Sub-Scalars

Typically, sub-scalars obtained from decomposition methods can be positive as well as negative. However, for efficiency reasons (specifically, to get the minimum length in the representation) the basic version of GLV-SAC only works for positive integers (see at the beginning of Section 3). In general, a straightforward solution during the scalar multiplication is to simply convert negative sub-scalars to positive and then negate the corresponding base P_i. Recoding to GLV-SAC can then be performed with Algorithm 1 as before. However, an issue with this approach arises when the precomputed table needs to be partitioned in more than one sub-table (see, for example, the case of 8-GLV in the previous subsection). If the bases P_i are negated depending on the sign of their corresponding sub-scalar then cheap applications of the GLV endomorphism cannot be conveniently applied to compute instances of the original sub-table.

Algorithm 3 Protected Recoding Algorithm for the Extended GLV-SAC Representation.

Input: m l-bit integers $k_j = (k^j_{l-1}, \ldots, k^j_0)_2$ and their corresponding signs $s_j \in \{-1, 1\}$ for $0 \leq j < m$, an odd "sign-aligner" $k_J \in \{k_j\}^m$, where $l = \lceil \log_2 r/m \rceil + 2$, m is the GLV dimension and r is the prime subgroup order.
Output: $(b^j_{l-1}, \ldots, b^j_0)_{\text{Ext-GLV-SAC}}$ for $0 \leq j < m$, where $b^J_i \in \{1, -1\}$, and $b^j_i \in \{0, b^J_i\}$ for $0 \leq j < m$ with $j \neq J$.

```
1: $b^J_{l-1} = 1$
2: for $i = 0$ to $(l-2)$ do
3:     $b^J_i = s_J \cdot (2k^J_{i+1} - 1)$
4: for $j = 0$ to $(m-1), j \neq J$ do
5:     for $i = 0$ to $(l-1)$ do
6:         $b^j_i = b^J_i \cdot k^j_0$
7:         $k_j = \lfloor k_j/2 \rfloor - s_j \cdot \lfloor b^j_i/2 \rfloor$
8: return $(b^j_{l-1}, \ldots, b^j_0)_{\text{Ext-GLV-SAC}}$ for $0 \leq j < m$.
```

To solve this problem, we present an extended version of the GLV-SAC representation that exhibits the following properties:

(i) The length of the digit representation of every sub-scalar k_j is fixed and given by $l = \lceil \log_2 r/m \rceil + 2$, where r is the prime subgroup order.

(ii) and (iii) are the same as the original GLV-SAC representation (see at the beginning of Section 3).

Thus, the only difference with the original GLV-SAC representation is the addition of one digit to the length, in order to deal with the sign. We present Algorithm 3 for the efficient recoding to this representation.

4 Modified LSB-Set Comb Method for Fixed-Base Scalar Multiplication

The side-channel protected comb methods by Feng et al. [13,14] and Hedabou et al. [22] used a simple version of the Lim-Lee's comb method that is restricted to only one table (see [28] for more details). Recently, Hamburg [19] precisely proposed to combine the original multi-table Lim-Lee's comb approach with Hedabou et al.'s SAB-set representation for improved performance. In this section, we follow a similar observation to extend Feng et al.'s LSB-set comb method with the use of multiple tables.

Let t be the bitlength of the prime subgroup order r. Assume that the binary representation of a scalar $k \in [1, r-1]$ is updated to $(k_{l-1}, k_{l-2}, \ldots, k_0)$ by padding with $(dw - t)$ zeros to the left for some window width w, where $l = dw$, $d = \lceil t/w \rceil$ and $w \geq 2$ (see Section 2.3). First, we note that the basic LSB-set comb method [13] requires additional operations in comparison with the original comb method when $w|t$. In [14], Feng et al. fixed this deficiency by proposing an MSB-set representation that recodes the most significant d bits to a nonzero representation, instead of the least significant d bits. In addition, MSB-set lifts the restriction to odd integers, inherent to the LSB-set representation. We show here that it is possible to modify the original LSB-set representation to fix the problem with the extra operations. Moreover, we easily solve the restriction to odd integers by exploiting the group order during scalar multiplication, and thus benefit from a somewhat simpler recoding algorithm.

As in [28], we add a parameter, referred to as v, that represents the number of tables to use. The modified LSB-set representation (denoted by mLSB-set) has the following properties

(i) Given a window width $w \geq 2$, a table parameter $v \geq 1$ and an odd integer $k \in [1, r-1]$, where r is the prime subgroup order, the digit-length of mLSB-set(k) is given by $l = dw$, where $d = ev$, $e = \lceil t/(wv) \rceil$ and $t = \lceil \log_2 r \rceil$.

(ii) The least significant d digits $(b_{d-1}, \ldots, b_0)$ are in signed nonzero form, i.e., $b_i \in \{1, -1\}$ for $0 \leq i < d$.

(iii) Remaining digits b_i for $d \leq i < l$ are expressed by a signed representation $(c, b_{l-1}, \ldots, b_d)$ s.t. $b_i \in \{0, b_{i \bmod d}\}$ and $c \in \{0, 1\}$. If $wv \nmid t$, c is always 0 and can be discarded.

Similarly to GLV-SAC, (i) and (ii) will enable a constant-time execution regardless of the value of k without having to appeal to masking for dealing with the identity element. Item (iii) will allow us to reduce the size of the precomputed table by a factor of 2, while minimizing the cost of precomputation. In our optimized setting using multiple tables, there is an additional carry bit c whenever $wv \mid t$. We remark that the latter reveals nothing since the appearance of the carry bit depends on public parameters. To deal with this extra bit during scalar multiplication we perform a correction with a precomputed value (see next section). Again, the condition that k should be odd enables the conversion of the least significant d bits to a signed nonzero representation using the equivalence $1 \equiv 1\bar{1}\ldots\bar{1}$. To deal with this restriction during scalar multiplication, we take advantage that our setting consists of a subgroup of prime order r. Hence, $-k \pmod{r} = r - k$ gives an odd result if k is even.

Algorithm 4 Protected Odd-Only Recoding Algorithm for the Modified LSB-Set Representation.

Input: An odd l-bit integer $k = (k_{l-1}, \ldots, k_0)_2$, window width $w \geq 2$ and table parameter $v \geq 1$, where $l = dw$, $d = ev$ and $e = \lceil \log_2 r/(wv) \rceil$.
Output: $(c, b_{l-1}, \ldots, b_0)_{m\text{LSB-set}}$, where $b_i \in \{1, -1\}$ for $0 \leq i < d$ and $b_i \in \{0, b_{i \bmod d}\}$ for $d \leq i \leq l-1$. If $wv|t$ then $c \in \{0, 1\}$, otherwise, $c = 0$ can be discarded.

```
1: $b_{d-1} = 1$
2: for $i = 0$ to $(d-2)$ do
3:     $b_i = 2k_{i+1} - 1$
4: $c = \lfloor k/2^d \rfloor$
5: for $i = d$ to $(l-1)$ do
6:     $b_i = b_{i \bmod d} \cdot c_0$
7:     $c = \lfloor c/2 \rfloor - \lfloor b_i/2 \rfloor$
8: return $(c, b_{l-1}, \ldots, b_0)_{m\text{LSB-set}}$.
```

An efficient algorithm to recode a given scalar to its mLSB-set representation is given in Algorithm 4. Conveniently, the recoding algorithm is simple and regular, exhibiting resistance to timing attacks. Note that, depending on the targeted platform, other implementation alternatives are possible. For example, Step 3 can be computed as $b_i = (k_{i+1} - 1) \,|\, 1$, where $|$ represents a logical OR operation.

Algorithm 5 Protected Fixed-Base Scalar Multiplication using the Modified LSB-Set Comb Method.

Input: A point $P \in E(\mathbb{F}_q)$ of prime order r, a scalar $k \in [1, r-1]$, window width $w \geq 2$, table parameter $v \geq 1$, where $d = ev$, $e = \lceil t/wv \rceil$ and $t = \lceil \log_2 r \rceil$.
Output: kP.

```
Precomputation stage:
1: Compute
   $P[u][v'] = 2^{ev'}(1 + u_0 2^d + \ldots + u_{w-2} 2^{(w-1)d})P$
   for all $0 \leq u < 2^{w-1}$ and $0 \leq v' < v$, where $u = (u_{w-2}, \ldots, u_0)_2$. If $wv \mid t$ then compute $2^{wd}P$.
Recoding stage:
2: even $= k \bmod 2$
3: if even $= 0$ then $k = r - k$
4: Pad $k$ with $(dw - t)$ zeros to the left and convert it to the $m$LSB-set representation using Algorithm 4
   s.t. $k = (c, b_{l-1}, \ldots, b_0)_{m\text{LSB-set}}$, 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
   $\mathbb{K}_{v',e'} = [K^{w-1}_{v',e'}, \ldots, K^2_{v',e'}, K^1_{v',e'}]$
   $\equiv |K^{w-1}_{v',e'} 2^{w-2} + \ldots + K^2_{v',e'} 2 + K^1_{v',e'}|$
   and digit-column signs $s_{v',e'} = K^0_{v',e'}$.
Evaluation stage:
5: Set
   $Q = s_{0,e-1} P[\mathbb{K}_{0,e-1}][0] + s_{1,e-1} P[\mathbb{K}_{1,e-1}][1]$
   $+ \ldots + s_{v-1,e-1} P[\mathbb{K}_{v-1,e-1}][v-1]$
6: for $i = e-2$ to $0$ do
7:     $Q = 2Q$
8:     $Q = Q + s_{0,i} P[\mathbb{K}_{0,i}][0] + s_{1,i} P[\mathbb{K}_{1,i}][1]$
           $+ \ldots + s_{v-1,i} P[\mathbb{K}_{v-1,i}][v-1]$
9: if ($wv \mid t \wedge c = 1$) then $Q = Q + 2^{wd}P$
10: if even $= 0$ then $Q = -Q$
11: return $Q$
```

4.1 Fixed-Base Scalar Multiplication using mLSB-Set

The new algorithm for computing fixed-base scalar multiplication using the modified LSB-set representation is shown as Algorithm 5. Disregarding c, the basic idea is to split $m\text{LSB-set}(k) = (c, b_{l-1}, \ldots, b_0)$ in consecutive d-digit parts $K^{w'}$, for $0 \leq w' < w$, and arrange them in matrix form from top to bottom, with the least significant d digits (i.e., K^0) at the top. Then, partition each $K^{w'}$ in v strings of e digits each, where $v \geq 1$ represents a suitably chosen number of tables (each table consists of v consecutive digit-columns). Thus, we have that $K^{w'}_{v',e'} = b_{dw'+ev'+e'}$, where $K^{w'}_{v',e'}$ denotes the e'-th digit in the v'-th string of a given $K^{w'}$. We then run a simultaneous multi-scalar multiplication scanning digit-columns from left to right, taking an entry from each table. With the mLSB-set recoding, every digit-column is nonzero by definition and has any of the possible combinations $[K^{w-1}_{v',e'}, \ldots, K^1_{v',e'}, K^0_{v',e'}]$, where $K^0_{v',e'} \in \{1, -1\}$, and $K^{w'}_{v',e'} \in \{0, K^0_{v',e'}\}$ for $1 \leq w' < w$. Since nonzero digits in the same column have the same sign, one only needs to precompute all the positive combinations $P[u][v'] = 2^{ev'}(1 + u_0 2^d + \ldots + u_{w-2} 2^{(w-1)d})P$ for all $0 \leq u < 2^{w-1}$ and $0 \leq v' < v$, where $u = (u_{w-2}, \ldots, u_0)_2$ and P is the base point. Assuming that negation of group elements is inexpensive in a given curve subgroup, negative values can be computed on-the-fly during the evaluation stage. Note that, in the fixed-base scenario, P is assumed to be known in advance and, hence, the precomputation can be computed off-line. Finally, if $wv \mid t$ for the chosen values for w and v, we need to precompute the extra point $2^{wd}P$ and apply a correction at the end of the evaluation stage (Step 10 of Alg. 5). A detailed proof is shown in Appendix A.

As can be seen, the main loop of Algorithm 5 computes kP using the regular pattern of one doubling and v additions. This regularity in the execution is the first requisite towards achieving protection against timing attacks and SSCA attacks such as SPA. Following previous recommendations, table accesses and conditional statements (e.g., Steps 3, 5, 8-10) should be performed in constant-time to guarantee protection against timing attacks. We also note that for the GLV-setting Algorithm 5 does not exploit endomorphisms. It is an open problem to derive a scalar multiplication method that exploits endomorphisms efficiently in the fixed-base case.

In the on-line computation, the method requires $e - 1 = \lceil \frac{t}{w \cdot v} \rceil - 1$ doublings and $ev - 1 = v \lceil \frac{t}{w \cdot v} \rceil - 1$ additions, using $v \cdot 2^{w-1}$ precomputed points. This cost should be increased in one addition using an extra precomputed point for the case in which $wv \mid t$. This cost is similar to Hamburg's method using the SAB-set representation [19]. In comparison, the SAB-set [22] and MSB-set [14] comb methods require approximately $(d-1)$ doublings and d additions using 2^{w-1} points, where $d = \lceil t/w \rceil$. For example, if we assume that one addition costs 1.3 doublings and $t = 256$ bits using 256 precomputed points, previous comb methods cost approximately 66 doublings (with $w = 9$), whereas the proposed method costs approximately 57 doublings (with $w = 8, v = 2$ optimal), injecting a 14% speedup.

Example 3. Let $w = 2$, $v = 2$ and $kP = 395P$. Using Algorithm 4 and assuming $t = 9$, the corresponding mLSB-set representation with fixed digit-length $l = 6 \cdot 2 = 12$, where $e = 3$ and $d = 6$, arranged in matrix form from top to bottom, is given by

$$\begin{bmatrix} K^0 \\ K^1 \end{bmatrix} \equiv \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \equiv \left[\begin{array}{ccc|ccc} 1 & \bar{1} & \bar{1} & 1 & \bar{1} & 1 \\ 1 & \bar{1} & \bar{1} & 0 & \bar{1} & 0 \end{array}\right]$$

Following Alg. 5, the digit columns are given by $\mathbb{K}_{0,2} = [0] = 0$, $\mathbb{K}_{0,1} = [\bar{1}] = 1$, $\mathbb{K}_{0,0} = [0] = 0$, $\mathbb{K}_{1,2} = [1] = 1$, $\mathbb{K}_{1,1} = [\bar{1}] = 1$ and $\mathbb{K}_{1,0} = [\bar{1}] = 1$, and their corresponding $s_{v',e'}$ are $s_{0,2} = 1$, $s_{0,1} = -1$, $s_{0,0} = 1$, $s_{1,2} = 1$, $s_{1,1} = -1$ and $s_{1,0} = -1$. Precomputed values $P[u][v']$ are given by

$P[0][0] = P$,
$P[1][0] = (1 + 64)P = 65P$
$P[0][1] = 8P$
$P[1][1] = 8 \cdot (1 + 64)P = 520P$

In the evaluation stage we first compute

$$\begin{aligned} Q &= s_{0,2}P[\mathbb{K}_{0,2}][0] + s_{1,2}P[\mathbb{K}_{1,2}][1] \\ &= P + 520P \\ &= 521P \end{aligned}$$

and then execute as shown in Table 2.

5 High-Speed Implementation on GLV-GLS Curves

In this section, we describe implementation aspects of the GLV-GLS curve `Ted127-glv4`. We present optimized algorithms for prime field, quadratic extension field and point arithmetic. We also present the technique of interleaving NEON and ARM-based multiprecision operations over $\mathbb{F}_{p^2}$. Although our techniques are especially tuned for the targeted curve, we remark that they can be adapted and exploited in other scenarios.

5.1 The Curve

For complete details about the 4-dimensional method using GLV-GLS curves, the reader is referred to [16] and [32]. We use the following GLV-GLS curve in twisted Edwards form [31], referred to as `Ted127-glv4`:

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

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 $2^{251} - (749)2^{128} - (1282451682958989391)2^{64} - 4923708382627145895$. 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 [31, 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 4-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

$$kP = k_1P + k_2\Phi(P) + k_3\Psi(P) + k_4\Psi\Phi(P),$$

with $\max_i(|k_i|) < C\, r^{1/4}$, where $C = 179$ [31]. Let $\zeta_8 = u/\sqrt{2}$ be a primitive 8th root of unity. The affine formulas for Φ and Ψ 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 $\Psi(x, y) = (\zeta_8 x^p, 1/y^p)$, 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 B.

Note that the curve `Ted127-glv4` has $a = -1$ (in the twisted Edwards equation; see [3]), corresponding to the most efficient set of formulas proposed by Hisil et al. [23]. Although GLV-GLS curves with suitably chosen parameters when transformed to twisted Edwards form offer roughly the same performance, as discussed in [31], there are certain differences in the cost of formulas for computing the endomorphisms Φ and Ψ. `Ted127-glv4` exhibits relatively efficient formulas for computing the endomorphisms in comparison with other GLV-GLS curves from [31]. 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 the next section for details). Also, since $p \equiv 3 \pmod 4$, $\beta = -1$ 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.

5.2 Field Arithmetic

For field inversion, we use the modular exponentiation $a^{p-2} \pmod p \equiv a^{-1}$ using a fixed and short

i	1	0
$2Q$	$1042P$	$914P$
$Q + s_{v',i}P[\mathbb{K}_{v',i}][v']$	$(1024 - 65 - 520)P = 457P$	$(914 + 1 - 520)P = 395P$

Table 2 Execution of the main loop of Algorithm 5 in Example 3.

addition chain. This method is simple to implement and inherently 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 $0 \le a, b < 2^{m+1}$. 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 $R = M_L + 2c\,M_H \le 2^{m+1}(2c+1) - 2$. 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. [8] 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 5.4.

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

Incomplete Reduction. Similar to [30], we exploit the form of the pseudo-Mersenne prime in combination with the incomplete reduction technique to speed up computations. We also mix incompletely reduced and completely reduced operands in novel ways.

Let $0 \le a < 2^m - c$ and $0 \le b < 2^m$. 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 $(m+1)^{\text{th}}$ bit of $a + b$. The result is correct 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 and b, such that $0 \le a < 2^m$ and $0 \le b < 2^m - c$. The standard field subtraction $(a - b) \bmod (2^m - c)$ described above will then produce an incompletely reduced result in the range $[0, 2^m - 1]$, since $a - b$ with $\mathit{borrow} = 0$ produces a result in the range $[0, 2^m - 1]$ and $a - b$ with $\mathit{borrow} = 1$ 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 for an addition preceding a subtraction. For example, this technique is exploited in the point doubling computation (see Steps 7-8 of Algorithm 11). 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. [8]. Again, let $0 \le a, b < 2^m - c$. Field addition is then computed as $((a + c) + b) - \mathit{carry} \cdot 2^m - (1 - \mathit{carry}) \cdot c$, where $\mathit{carry} = 0$ if $a + b + c < 2^m$, otherwise $\mathit{carry} = 1$. Similar to previous cases, the subtraction with $\mathit{carry} \cdot 2^m$ can be efficiently carried out by clearing the $(m+1)^{\text{th}}$ bit in $(a+c)+b$. As discussed above, this efficient computation is also advantageously exploited in the modular reduction for multiplication and squaring.

5.3 Quadratic Extension Field Arithmetic

For the remainder, we use the following notation: (i) I, M, S, A and R represent inversion, multiplication, squaring, addition and modular reduction over $\mathbb{F}_p$, 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 127-bit prime. The details for the x64 implementation are shown in Algorithm 6. 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 *rdcn*= *FALSE* in Alg. 6) 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 *rdcn*= *TRUE*). Squaring over $\mathbb{F}_{p^2}$ is computed using the complex method. The details for the x64 implementation are shown in Algorithm 7. 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.

Algorithm 6 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 $0 \le a_0, a_1, b_0, b_1 \le 2^{127} - 1, p = 2^{127} - c, c$ 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 $rdcn = TRUE$, 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 $rdcn = TRUE$, then $c_1 \leftarrow C_1 \bmod p$             $[0, p>$
12: return if $rdcn = TRUE$ then $a \cdot b = (c_0 + c_1 i)$, else
    $a \cdot b = (C_0 + C_1 i)$ .
```

Algorithm 7 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 $0 \le a_0, a_1 \le p - 1$, $p = 2^{127} - c, c$ 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 $a^2 = (c_0 + c_1 i)$.
```

5.4 Extension Field Arithmetic on ARM: Efficient Interleaving of ARM-Based and NEON-Based Multiprecision Operations

The potential performance gain when interleaving ARM and NEON operations is well-known. This feature was exploited in [7] to speed up the Salsa20 stream cipher. On the other hand, Sánchez and Rodríguez-Henríquez [38] showed how to take advantage of NEON instructions to perform independent multiplications in operations over $\mathbb{F}_{p^2}$. In the following, we go a step further and show how to exploit the increasingly efficient capacity of modern ARM processors for executing ARM and NEON instructions "simultaneously" to implement multiprecision operations, such as multiplication, squaring and modular reduction, over

$\mathbb{F}_{p^2}$. 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 vice versa. Note that loading or 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 and squaring 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:

- `double_mul_neonarm`: a double integer multiplier detailed in Algorithm 8, which interleaves a single 128-bit multiplication using NEON and a single 128-bit multiplication using ARM.
- `triple_mul_neonarm`: a triple integer multiplier detailed in Algorithm 9, which interleaves two single 128-bit multiplication using NEON and one single 128-bit multiplication using ARM.
- `double_red_neonarm`: a double modular reduction algorithm detailed in Algorithm 10, that interleaves a modular reduction using ARM and a modular reduction using NEON.

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 [29], which add the result of a multiplication or shift operation to the destination register in one single operation, reducing code size.

To validate the efficiency of our approach, we compared the interleaved algorithms above with standard implementations using only NEON or only ARM instructions. In all the cases, we observed a reduction of costs in favor of our novel interleaved ARM/NEON implementations (see Section 6 for benchmark results).

`Triple_mul_neonarm` fits nicely in the computation of multiplication over $\mathbb{F}_{p^2}$, since this operation requires three 128-bit integer multiplications (Steps 1, 2 and 8 of Alg. 6). Similarly, 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 Alg. 7). Finally, for each case we can efficiently use a `double_red_neonarm`. The final algorithms for ARM are shown as Algorithms 13 and 14 in Appendix C.

5.5 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. [23], corresponding to the case $a = -1$, that uses a combination of homogeneous projective coordinates $(X : Y : Z)$ and the 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 11 and 12, respectively. In these algorithms, we extensively exploit incomplete reduction (denoted with $\oplus, \ominus$), following the details given in Section 5.2. 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 standard 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 11 and 12. For this case, precomputed points are more efficiently represented as $(X+Y, Y-X, 2Z, 2T)$ (corresponding to setting *EXT_COORD*= *TRUE* in Alg. 12), 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 three additions and one multiplication per iteration (Steps 2, 8 and 11 of Alg. 12), 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 $(x + y, y - x, 2t)$, mixed addition costs $7m + 7a$. In the fixed/variable-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 $8m+6a$;

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

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

```
1: $(F, G) \leftarrow (0, 0)$
2: for $i = 0$ to $1$ do
3:     $(C_0, C_1, C_2) \leftarrow (0, 0, 0)$
4:     for $j = 0$ to $3$ do
5:         $(C_0, F_{i+j}, C_1, F_{i+j+2}) \leftarrow (F_{i+j} + a_i b_j + C_0, F_{i+j+2} + a_{i+2} b_j + C_1)$    {done by NEON}
6:     for $j = 0$ to $3$ do
7:         $(C_2, G_{i+j}) \leftarrow G_{i+j} + c_j d_i + C_2$    {done by ARM}
8:     $(F_{i+4}, F_{i+6}, G_{i+4}) \leftarrow (F_{i+4} + C_0, C_1, C_2)$
9: for $i = 2$ to $3$ do
10:    for $j = 0$ to $3$ do
11:        $(C_2, G_{i+j}) \leftarrow G_{i+j} + c_j d_i + C_2$    {done by ARM}
12:    $G_{i+4} \leftarrow C_2$
13: return $(F, G)$
```

Algorithm 9 Triple 128-bit integer product with ARM and NEON interleaved (`triple_mul_neonarm`).

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

```
1: $(F, G, H) \leftarrow (0, 0, 0)$
2: for $i = 0$ to $3$ do
3:     $(C_0, C_1, C_2) \leftarrow (0, 0, 0)$
4:     for $j = 0$ to $3$ do
5:         $(C_0, F_{i+j}, C_1, G_{i+j}) \leftarrow (F_{i+j} + a_j b_i + C_0, G_{i+j} + c_j d_i + C_1)$    {done by NEON}
6:     for $j = 0$ to $3$ do
7:         $(C_2, H_{i+j}) \leftarrow H_{i+j} + e_j f_i + C_2$    {done by ARM}
8:     $(F_{i+4}, G_{i+4}, H_{i+4}) \leftarrow (C_0, C_1, C_2)$
9: return $(F, G, H)$
```

Algorithm 10 Double modular reduction with ARM and NEON interleaved (`double_red_neonarm`).

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

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

Algorithm 11 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: $T_a \leftarrow T_a \times Y_1$            $(2X_1 Y_1)$
12: $X_2 \leftarrow T_a \times t_1$            $(X_2 = 2X_1 Y_1 [2Z_1^2 - (Y_1^2 - X_1^2)])$
13: return $2P = (X_2, Y_2, Z_2)$ and $\{T_a, T_b\}$ such that $T_2 = T_a \cdot T_b$.
```

Algorithm 12 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 $t_2 \leftarrow T_1 \times 2Z_2$    $(2T_1Z_2)$
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 $Y_3 = Y_2 - X_2$, 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 $X_3 = X_2 + Y_2$, 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 $T_3 = T_a \cdot T_b$.
```

whereas points corresponding to the fixed base can again be represented as (x, y) or as $(x+y, y-x, 2t)$, following the same trade-offs and costs for mixed addition discussed above. These options are also evaluated in Section 6.

6 Performance Analysis and Experimental Results

In this section, we carry out the performance analysis of the proposed GLV-SAC representation for GLV-based variable-base scalar multiplication and the mLSB-set method for fixed-base scalar multiplication, and present benchmark results of our constant-time implementations of curve `Ted127-glv4` on x64 and ARM platforms. We also assess the performance improvement obtained with the proposed ARM/NEON interleaving technique. 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 (equipped with a NEON vector unit). The x64 implementation, compatible with both Windows and Linux OS, was compiled using Microsoft Visual Studio 2012 in the case of Windows (Microsoft Windows 8 OS) and GNU GCC v4.7.3 in the case of Linux (Ubuntu 14.04 LTS). In our experiments, we turned off Intel's Hyper-Threading and Turbo Boost technologies; 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, we present timings for all the fundamental operations of scalar multiplication in Table 3. Implementation details for the quadratic extension field operations and point operations over $\mathbb{F}_{p^2}$ can be found in Section 5. "IR" stands for incomplete reduction and "extended" represents the use of the extended coordinates $(X + Y, Y - X, 2Z, 2T)$ to represent precomputed points. The four-dimensional decomposition of the scalar follows [31]. In particular, a scalar k is decomposed in smaller k_i s.t. $\max(|k_i|) < C\, r^{1/4}$ for $0 \leq i \leq 3$, where r is the 251-bit prime order and $C = 179$ for our case (see §5.1). In practice, however, we have found that the bitlength of k_i is at most 63 bits for our targeted curve. The decomposition can be performed as a linear transformation by computing $k_i = \sum_{j=0}^{3} \texttt{round}(S_j k) \cdot M_{i,j}$ for $0 \leq i < 4$, where $M_{i,j}$ and S_j are integer constants. We truncate operands in the `round()` operation, adding enough precision to avoid loss of data. Thus, the computation involves a few multi-precision integer operations exhibiting constant-time execution.

Table 3 Cost (in cycles) of basic operations on curve `Ted127-glv4`.

Operation		ARM Cortex-A9	ARM Cortex-A15	Intel Sandy Bridge (Linux / Win8)	Intel Ivy Bridge (Linux / Win8)
$\mathbb{F}_{p^2}$	ADD with IR	20	19	12 / 12	13 / 12
	SUB	39	37	13 / 12	12 / 12
	SQR	223	141	57 / 59	55 / 56
	MUL	339	185	71 / 78	69 / 75
	INV	13,390	9,675	6,100 / 6,060	5,880 / 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	84 / 83	80 / 79
	GLV-SAC recoding	1,236	873	500 / 482	502 / 482
	4-GLV decomposition	756	430	304 / 305	295 / 290

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

Variable-Base Scenario. Based on Alg. 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 (denoted by $LUT8$) in the evaluation stage; and 1 inversion and 2 multiplications over $\mathbb{F}_{p^2}$ for converting the final result to affine:

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

This operation count does not take into account other additional computations, such as the recoding to the GLV-SAC representation or the decomposition to 4-GLV, which are relatively inexpensive (see Table 3).

Compared to [31], which uses a method based on the regular windowed recoding [34], the GLV-SAC 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 during scalar multiplication, whereas the method in [31] requires 36 precomputed points, which involve 68 protected table lookups over 9 points. For example, this represents in practice a 17% speedup in the computation and a 78% reduction in the memory consumption of precomputation on curve `Ted127-glv4`.

Fixed-Base Scenario. In this case, we analyze costs when using the mLSB-set comb method (Algorithm 5) with 32, 64, 128, 256 and 512 precomputed points. Recalling Section 4, and since $t = 251$, the method costs $\lceil \frac{251}{w \cdot v} \rceil - 1$ doublings and $v \lceil \frac{251}{w \cdot v} \rceil - 1$ additions 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 6, Appendix D, 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. As can be seen, in most cases each precomputation representation is optimal for specific storage values in the targeted platforms. For large tables, however, extended affine coordinates achieve better results.

Fixed/Variable-Base Double Scalar Scenario. Since this operation does not need to run in constant time, it can be computed using the standard width-w non-adjacent form (w-NAF) with interleaving [17,33]. In the setting of m-dimensional GLV, the idea is to split the two scalars in two sets of m sub-scalars with maximum bitlength $l = \lceil \log_2 r/m \rceil$ each, convert them to w-NAF and run a multi-exponentiation. Thus, the cost is given by $m \cdot (\frac{l}{w_1+1})$ mixed additions, $m \cdot (\frac{l}{w_2+1}) - 1$ 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 fixed base and w_2 is the window size for the variable base. 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. Moreover, it is possible to choose different window sizes for each sub-scalar, giving more flexibility in the selection of the optimal number of precomputations. In this direction, we analyze the cost of `Ted127-glv4` for different values of $w_{1,j}$

(corresponding to each of the j sub-scalars of the fixed base, where $0 \leq j < 4$), using the optimal value $w_2 = 4$ for the sub-scalars of the variable base. This value for w_2 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 $\left(\sum_{j=0}^{3} \frac{63}{w_{1,j}+1}\right) + 3$ mixed additions in the evaluation stage using sixteen "ephemeral" points and $\sum_{j=0}^{3} 2^{w_{1,j}-2}$ "permanent" points; 2 doublings, 6 additions, 8 Ψ endomorphisms and one Φ endomorphism in the online precomputation stage; and 1 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 7, Appendix E, 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.

6.1 Results

Table 4 includes our benchmark results for all the core scalar multiplication operations: variable-base, fixed-base and fixed/variable-base double scalar scenario. In Table 5 we compare them with other implementations in the literature for the variable-base case. The reader should note that we compare timings for constant-time implementations only. Hence, results are not directly comparable with an analysis based on *unprotected* implementations [16,15].

The results for the representative variable-base scenario set a new speed record for protected elliptic curve scalar multiplication on several x64 and ARM processors. In comparison to the previously fastest x64 implementation by Longa and Sica [31], which runs in 137,000 cycles, the presented result injects a cost reduction of about 30% on a Sandy Bridge machine. Our results are more than 2 times faster than Bernstein et al.'s implementation using a Montgomery curve over $\mathbb{F}_p$ [5] on the targeted x64 processors. In comparison with curve-based implementations on genus 2 curves or binary curves, we observe that our results are between 24%-26% faster than the genus 2 implementation by Bos et al. [8], and between 19%-24% faster than the implementation by Oliveira et al. [35] based on a binary GLS curve using the 2-GLV method[1]. Only the recent implementation by Bernstein et al. [4], which uses the same genus 2 Kummer surface employed by Bos et al. [8], is able to achieve a performance that is comparable to this work, with a result that is slightly slower on the Intel Ivy Bridge processor. In addition, the applicability of the Kummer surface is essentially restricted to ECDH; e.g., it cannot be used directly for signature schemes and its performance is poor in ECDHE when precomputations (via fixed-base scalar multiplication) can be exploited. Our results also demonstrate that the proposed techniques enable a significant reduction of 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 (Windows OS), which is only 9% faster than our protected version. In the case of ARM, our implementation of variable-base scalar multiplication on curve `Ted127-glv4` is 27% and 32% faster than Bernstein and Schwabe's [6] and Hamburg's [19] Montgomery curve implementations (respect.) on a Cortex-A9 processor. Note, however, that comparisons on ARM are particularly difficult. The implementation of [7] was originally optimized for Cortex-A8, and the implementation of [19] does not exploit NEON.

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

- Hamburg [19] 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 51K cycles and 204K cycles, respectively.
- Hamburg [19] 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 119K cycles and 495K cycles, respectively, using the same table size.

Finally, when assessing the improvement obtained with the ARM/NEON interleaving technique on the Cortex-A9 processor, we observed

[1] In the case of *unprotected* software on x64, Oliveira et al. [35] hold the current speed record with 72,000 cycles on an Intel Sandy Bridge. Their protected version is significantly more costly and runs in about 115,000 cycles.

Table 4 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 10^3 cycles.

Scalar Multiplication	Parameters	ARM Cortex-A9	ARM Cortex-A15	Intel Sandy Bridge (Linux / Win8)	Intel Ivy Bridge (Linux / Win8)
kP, variable base	8 points, 1KB, extended	417	244	92 / 96	89 / 92
kP, fixed base	$v = 4, w = 5$, 64 points, 6KB, ext.	204	116	51 / 54	49 / 52
	$v = 4, w = 6$, 128 points, 12KB, ext.	181	106	47 / 50	45 / 49
	$v = 8, w = 6$, 256 points, 24KB, ext.	172	100	45 / 48	43 / 46
$kP + lQ$	$w_2 = 3$, 8 points, 768 bytes, extended	560	321	131 / 136	126 / 130
	$w_2 = 5$, 32 points, 3KB, extended	495	285	119 / 123	115 / 118
	$w_2 = 7$, 128 points, 12KB, extended	463	266	112 / 116	108 / 111

Table 5 Cost (in 10^3 cycles) of implementations of variable-base scalar multiplication with protection against timing-type side-channel attacks at approximately 128-bit security level. Results are approximated to the nearest 10^3 cycles. Only results for Linux are shown for the Intel platforms.

Work (Curve)	ARM Cortex-A9	ARM Cortex-A15	Intel Sandy Bridge	Intel Ivy Bridge
`Ted127-glv4` (this work)	417	244	92	89
`Ted127-glv4`, Longa-Sica [31]	-	-	137 (*)	-
Montgomery curve $E/\mathbb{F}_p$, Hamburg [19]	616	-	153	-
`Curve25519`, Bernstein et al. [5,7]	568 (**)	-	194 (**)	183 (**)
Binary GLS $E/\mathbb{F}_{2^{254}}$, Oliveira et al. [35]	-	-	115	113
Genus 2 Kummer $C/\mathbb{F}_p$, Bos et al. [9]	-	-	126 (**)	117
Genus 2 Kummer $C/\mathbb{F}_p$, Bernstein et al. [4]	-	-	92	91

(*) Running on Windows 7.
(**) Source: eBACS [6].

speedups close to 17% and 24% in comparison with implementations exploiting only ARM or NEON instructions, respectively. Remarkably, for the same figures on the Cortex-A15, we observed speedups in the order of 34% and 35%, respectively. These experimental results confirm the significant performance improvement enabled by the proposed technique, which exploits the increasing capacity of the latest ARM processors for parallelizing ARM and NEON instructions.

Acknowledgements We would like to thank Joppe Bos, Craig Costello, Francisco Rodríguez-Henríquez and the reviewers for their useful comments that helped us improve the quality of this work. Also, we would like to thank Francisco Rodríguez-Henríquez 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 K.G. Paterson, editor, *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, C. Chuengsatiansup, T. Lange, and P. Schwabe. Kummer strikes back: new DH speed records. In *Cryptology ePrint Archive, Report 2014/134*, 2014. Available at: `http://eprint.iacr.org/2014/134`.
5. D. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. In B. Preneel and T. Takagi, editors, *Proceedings of CHES 2011*, volume 6917 of *LNCS*, pages 124–142. Springer, 2011.
6. D. Bernstein and T. Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems, accessed on Dec 12, 2013. `http://bench.cr.yp.to/results-dh.html`.
7. D. Bernstein and P. Schwabe. NEON crypto. In E. Prouff and P.R. Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 320–339. Springer, 2012.
8. J.W. Bos, C. Costello, H. Hisil, and K. Lauter. Fast cryptography in genus 2. In T. Johansson and P.Q. Nguyen, editors, *Advances in Crytology - EUROCRYPT*, volume 7881 of *LNCS*, pages 194–210. Springer, 2013.
9. J.W. Bos, C. Costello, H. Hisil, and K. Lauter. High-performance scalar multiplication using

8-dimensional GLV/GLS decomposition. In G. Bertoni and J.-S. Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, volume 8086 of *LNCS*, pages 331–348. Springer, 2013.

10. J.W. Bos, C. Costello, P. Longa, and M. Naehrig. Selecting elliptic curves for cryptography: An efficiency and security analysis. In *Cryptology ePrint Archive, Report 2014/130*, 2014. Available at: `http://eprint.iacr.org/2014/130`.
11. 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.
12. A. Faz-Hernández, P. Longa, and A.H. Sánchez. Efficient and secure algorithms for GLV-based scalar multiplication and their implementation on GLV-GLS curves. In J. Benaloh, editor, *Topics in Cryptology - CT-RSA 2014*, volume 8366, pages 1–27. Springer, 2014.
13. 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`.
14. 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.
15. 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.
16. 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.
17. 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.
18. A. Guillevic and S. Ionica. Four dimensional GLV via the Weil restriction. In K. Sako and P. Sarkar, editors, *Advances in Cryptology - ASIACRYPT*, volume 8269 of *LNCS*, pages 79–96. Springer, 2013.
19. M. Hamburg. Fast and compact elliptic-curve cryptography. In *Cryptology ePrint Archive, Report 2012/309*, 2012. Available at: `http://eprint.iacr.org/2012/309`.
20. D. Hankerson, K. Karabina, and A. Menezes. Analyzing the Galbraith-Lin-Scott point multiplication method for elliptic curves over binary fields. *IEEE Trans. Computers*, 58(10):1411–1420, 2009.
21. D. Hankerson, A. Menezes, and S. Vanstone. *Guide to elliptic curve cryptography.* Springer Verlag, 2004.
22. 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.
23. 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.
24. Z. Hu, P. Longa, and M. Xu. Implementing 4-dimensional GLV method on GLS elliptic curves with j-invariant 0. *Designs, Codes and Cryptography*, 63(3):331–343, 2012. Available at: `http://eprint.iacr.org/2011/315`.
25. M. Joye and M. Tunstall. Exponent recoding and regular exponentiation algorithms. In M. Joye, editor, *Proceedings of Africacrypt 2003*, volume 5580 of *LNCS*, pages 334–349. Springer, 2009.
26. 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.
27. 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.
28. 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.
29. ARM Limited. ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition, 2012.
30. 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.
31. 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.
32. P. Longa and F. Sica. Four-dimensional Gallant-Lambert-Vanstone scalar multiplication. *Journal of Cryptology*, 27(2):248–283, 2014.
33. 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.
34. K. Okeya and T. Takagi. The width-w NAF method provides small memory and fast elliptic curve scalars multiplications against side-channel attacks. In M. Joye, editor, *Proceedings of CT-RSA 2003*, volume 2612 of *LNCS*, pages 328–342. Springer, 2003.
35. T. Oliveira, J. López, D.F. Aranha, and F. Rodríguez-Henríquez. Lambda coordinates for binary elliptic curves. In G. Bertoni and J.-S. Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, volume 8086 of *LNCS*, pages 311–330. Springer, 2013.
36. 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.
37. Microsoft Research. MSR Elliptic Curve Cryptography Library (MSR ECCLib), 2014. Available at: `http://research.microsoft.com/en-us/projects/nums`.
38. A.H. Sánchez and F. Rodríguez-Henríquez. NEON implementation of an attribute-based encryption

scheme. In M. Jacobson, M. Locasto, P. Mohassel, and R. Safavi-Naini, editors, *International Conference on Applied Cryptography and Network Security - ACNS 2013*, volume 7954 of *LNCS*, pages 322–338. Springer, 2013.

39. B. Smith. Families of fast elliptic curves from $\mathbb{Q}$-curves. In K. Sako and P. Sarkar, editors, *Advances in Cryptology - ASIACRYPT*, volume 8269 of *LNCS*, pages 61–78. Springer, 2013.
40. 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.
41. 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.
42. S.-M. Yen and M. Joye. Checking before output may not be enough against fault- based cryptanalysis. *IEEE Trans. Computers*, 49(9):967–970, 2000.
43. S.-M. Yen, S. Kim, S. Lim, and S.-J. Moon. A countermeasure against one physical cryptanalysis may benefit another attack. In K. Kim, editor, *Information Security and Cryptology - ICISC 2001*, volume 2288 of *Lecture Notes in Computer Science*, pages 414–427. Springer, 2002.

A Comb Method using the Modified LSB-Set Representation

Let t be the bitlength of the prime subgroup order r. Assume that $k \in [1, r-1]$ is partitioned in w consecutive parts of d digits each, and each part is partitioned in v strings of e digits each, padding k with $(dw - t)$ zeros to the left, where $l = dw, d = ev$ and $e = \lceil t/wv \rceil$. The modified LSB-set representation of k is given by

$$k = c \cdot 2^l + \sum_{i=0}^{l-1} b_i 2^i \equiv (c, b_{l-1}, \dots, b_0)_{m\text{LSB-set}}, \quad (2)$$

where $b_i \in \{1, -1\}$ for $0 \le i < d$, and $b_i \in \{0, b_{i \bmod d}\}$ for $d \le i \le l-1$. If $wv \mid t$ then the carry bit $c \in \{0, 1\}$. Otherwise, c is always zero. Disregarding the carry bit, rewrite the representation (2) in matrix form as follows

$$\begin{aligned} k' &= k - c \cdot 2^l \\ &\equiv \begin{bmatrix} K^0 \\ \vdots \\ K^{w'} \\ \vdots \\ K^{w-1} \end{bmatrix} \equiv \begin{bmatrix} K^0_{v-1} & \cdots & K^0_{v'} & \cdots & K^0_0 \\ \vdots & & \vdots & & \vdots \\ K^{w'}_{v-1} & \cdots & K^{w'}_{v'} & \cdots & K^{w'}_0 \\ \vdots & & \vdots & & \vdots \\ K^{w-1}_{v-1} & \cdots & K^{w-1}_{v'} & \cdots & K^{w-1}_0 \end{bmatrix} \\ &\equiv \sum_{w'=0}^{w-1} K^{w'} 2^{dw'} \end{aligned} \quad (3)$$

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 $K^{w'}_{v',e'}$, such that $K^{w'}_{v',e'} = b_{dw'+ev'+e'}$. Then, to compute the scalar multiplication we have

$$\begin{aligned} k'P &= \sum_{w'=0}^{w-1} K^{w'} 2^{dw'} P = \sum_{w'=0}^{w-1} \sum_{v'=0}^{v-1} K^{w'}_{v'} 2^{ev'} 2^{dw'} P \\ &= \sum_{w'=0}^{w-1} \sum_{v'=0}^{v-1} \sum_{e'=0}^{e-1} K^{w'}_{v',e'} 2^{e'} 2^{ev'} 2^{dw'} P. \end{aligned} \quad (4)$$

Assuming that $P[w'] = 2^{dw'}P$ for $0 \le w' \le w-1$, then

$$\begin{aligned} k'P &= \sum_{e'=0}^{e-1} 2^{e'} \left(\sum_{v'=0}^{v-1} \sum_{w'=0}^{w-1} K^{w'}_{v',e'} 2^{ev'} P[w'] \right) \\ &= \sum_{e'=0}^{e-1} 2^{e'} \left(\sum_{v'=0}^{v-1} 2^{ev'} \left(K^0_{v',e'} P[0] \right.\right. \\ &\quad \left.\left. + \sum_{w'=1}^{w-1} K^{w'}_{v',e'} P[w'] \right) \right). \end{aligned} \quad (5)$$

Recall that by definition of the mLSB-set representation $K^0_{v',e'} \in \{1, -1\}$ and $K^{w'}_{v',e'} \in \{0, K^0_{v',e'}\}$ for $1 \le w' \le w-1$, for a given pair of indices (v', e'). Assume that the following values are precomputed for all $0 \le u < 2^{w-1}$ and $0 \le v' < v$

$$\begin{aligned} P[u][v'] &= 2^{ev'}(P[0] + u_0 P[1] + \cdots + u_{w-2} P[w-1]) \\ &= 2^{ev'}(1 + u_0 2^d + \ldots + u_{w-2} 2^{(w-1)d}) P, \end{aligned} \quad (6)$$

where $u = (u_{w-2}, \dots, u_0)_2$. Then, $k'P$ can be rewritten as

$$k'P = \sum_{e'=0}^{e-1} 2^{e'} \left(\sum_{v'=0}^{v-1} s_{v',e'} P[\mathbb{K}_{v',e'}][v'] \right), \quad (7)$$

where digit-columns $\mathbb{K}_{v',e'} = [K^{w-1}_{v',e'}, \dots, K^2_{v',e'}, K^1_{v',e'}]$ $\equiv |K^{w-1}_{v',e'} 2^{w-2} + \ldots + K^2_{v',e'} 2 + K^1_{v',e'}|$, and the sign $s_{v',e'} = K^0_{v',e'} \in \{1, -1\}$.

Based on equation (7), $k'P$ can be computed from left-to-right using precomputed points (6) together with a variant of the double-and-add algorithm (see Algorithm 5). The final result is obtained after a final correction computing $kP = k'P + c \cdot 2^{wd}P$ using the precomputed value $2^{wd}P$.

B 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 $\zeta_8 = u/\sqrt{2}$ be a primitive 8th root of unity. Then, we can compute $\Phi(P) = (X_2, Y_2, Z_2, T_2)$ as follows

$$\begin{aligned} X_2 &= -X_1 \left(\alpha Y_1^2 + \theta Z_1^2 \right) \left[\mu Y_1^2 - \phi Z_1^2 \right] \\ Y_2 &= 2Y_1 Z_1^2 \left[\phi Y_1^2 + \gamma Z_1^2 \right] \\ Z_2 &= 2Y_1 Z_1^2 \left[\mu Y_1^2 - \phi Z_1^2 \right] \\ T_2 &= -X_1 \left(\alpha Y_1^2 + \theta Z_1^2 \right) \left[\phi Y_1^2 + \gamma Z_1^2 \right] \end{aligned}$$

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

$$\begin{aligned} \zeta_8 &= 1 + Ai, \quad \mu = (A-1) + (A+1)i, \quad \theta = A + Bi, \\ \alpha &= A + 2i, \quad \gamma = (A+1) + (A-1)i, \quad \phi = B + 1 + i, \end{aligned}$$

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

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

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 $Z_1 = 1$.

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

Algorithms targeting ARM platforms for multiplication and squaring over $\mathbb{F}_{p^2}$, with $p = 2^{127} - c$, are detailed by Algorithms 13 and 14, respectively. These algorithms exploit functions interleaving ARM/NEON-based operations, namely `double_mul_neonarm`, `triple_mul_neonarm` and `double_red_neonarm`, which are detailed in Algorithms 8, 9 and 10, respectively.

D Cost of Fixed-Base Scalar Multiplication using the *m*LSB-Set Comb Method

In Table 6, we present estimated costs in terms of multiplications over $\mathbb{F}_{p^2}$ per bit for fixed-base scalar multiplication on curve `Ted127-glv4` using the mLSB-set comb method (Algorithm 5). Precomputed points are stored as (x, y) coordinates ("affine") or as $(x+y, y-x, 2t)$ coordinates ("extended").

E Cost of Fixed/Variable-Base Double Scalar Multiplication on Curve `Ted127-glv4` using *w*NAF with Interleaving

In Table 7,, we present estimated costs in terms of multiplications over $\mathbb{F}_{p^2}$ per bit for fixed/variable-base double scalar multiplication on curve `Ted127-glv4` using w-NAF 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 $w_{1,j}$ 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 $w_2 = 4$, corresponding to the use of 16 precomputed points (see §6).

Algorithm 13 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: $(T_0, T_1, T_2) \leftarrow \texttt{triple\_mul\_neonarm}(a_0, b_0, a_1, b_1, t_0, t_1)$   $[0, 2^{256} >$
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 $(c_0, c_1) \leftarrow \texttt{double\_red\_neonarm}(C_0, C_1)$   $[0, p >$
```

Algorithm 14 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: $(C_0, C_1) \leftarrow \texttt{double\_mul\_neonarm}(t_0, t_1, t_2, a_1)$   $[0, p^2 >$
5: return $a^2 = \texttt{double\_red\_neonarm}(C_0, C_1)$                   $[0, p >$
```

Table 6 Cost (in $\mathbb{F}_{p^2}$ multiplications per bit) of fixed-base scalar multiplication using the mLSB-set representation 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	affine	2.88	3.24	3.05	3.08
2, 5, 32 points, 2KB		**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	extended	2.76	3.11	2.92	2.94
2, 5, 32 points, 3KB		**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	affine	2.33	2.63	2.46	2.49
4, 5, 64 points, 4KB		**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	extended	2.21	2.50	2.33	2.35
4, 5, 64 points, 6KB		**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	affine	2.26	2.60	2.40	2.43
4, 6, 128 points, 8KB		**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	extended	2.25	2.60	2.37	2.40
4, 6, 128 points, 12KB		**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	affine	2.02	2.34	2.14	2.18
8, 6, 256 points, 16KB		**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	extended	2.02	2.34	2.12	2.15
8, 6, 256 points, 24KB		**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	affine	1.92	2.22	2.03	2.06
16, 6, 512 points, 32KB		**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	extended	1.91	2.22	2.00	2.04
16, 6, 512 points, 48KB		**1.75**	**1.97**	**1.82**	**1.85**
32, 5, 512 points, 48KB		1.83	2.03	1.91	1.93

Table 7 Cost (in $\mathbb{F}_{p^2}$ multiplications per bit) of fixed/variable-base double scalar multiplication on curve `Ted127-glv4` using w-NAF with interleaving, $w_2 = 4$, 16 "ephemeral" precomputed points.

$w_{1,j}$, # 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**