FourℚNEON: Faster Elliptic Curve Scalar Multiplications on ARM Processors

Patrick Longa(✉)

Microsoft Research, Redmond, USA
plonga@microsoft.com

Abstract. We present a high-speed, high-security implementation of the recently proposed elliptic curve Fourℚ (ASIACRYPT 2015) for 32-bit ARM processors with NEON support. Exploiting the versatile and compact arithmetic of this curve, we design a vectorized implementation that achieves high-performance across a large variety of ARM platforms. Our software is fully protected against timing and cache attacks, and showcases the impressive speed of Fourℚ when compared with other curve-based alternatives. For example, one single variable-base scalar multiplication is computed in about 235,000 Cortex-A8 cycles or 132,000 Cortex-A15 cycles which, compared to the results of the fastest genus 2 Kummer and Curve25519 implementations on the same platforms, offer speedups between 1.3x–1.7x and between 2.1x–2.4x, respectively. In comparison with the NIST standard curve K-283, we achieve speedups above 4x and 5.5x.

Keywords: Elliptic curves · Fourℚ · ARM · NEON · Vectorization · Efficient software implementation · Constant-time.

1 Introduction

In 2013, ARM surpassed the 50 billion mark of processors shipped worldwide, consolidating its hegemony as the most widely used architecture in terms of quantity [22]. One of the main drivers of this success has been the explosive growth of the mobile market, for which the Cortex-A and Cortex-M architectures based on the ARMv7 instruction set became key technologies. In particular, the Cortex-A series include powerful yet power-efficient processors that have successfully hit the smartphone/tablet/wearable mass market. For example, Cortex-A7 based SOCs power the Samsung Gear S2 (2015) smartwatch and the Microsoft Lumia 650 (2016) smartphone; Cortex-A8 and Cortex-A9 cores can be found in the Motorola Moto 360 (2014) smartwatch and the Samsung Galaxy Light (2013) smartphone, respectively; and Cortex-A15/Cortex-A7 (big.LITTLE) based SOCs power the Samsung Galaxy S5 (2014) and the Samsung Galaxy A8 (2015) smartphones. Many of these Cortex-A microarchitectures come equipped with a NEON engine, which provides advanced 128-bit Single Instruction Multiple Data (SIMD) vector instructions. Thus, these low-power

RISC-based ARM platforms with NEON support have become an attractive platform for deploying and optimizing cryptographic computations.

Costello and Longa [12] recently proposed a highly efficient elliptic curve, dubbed Four$\mathbb{Q}$, that provides around 128 bits of security and enables the fastest curve-based scalar multiplications on x64 software platforms by combining a four-dimensional decomposition based on endomorphisms [16], the fastest twisted Edwards formulas [18], and the efficient yet compact Mersenne prime $p = 2^{127} - 1$. In summary, the results from [12] show that, when computing a single variable-base scalar multiplication, Four$\mathbb{Q}$ is more than 5 times faster than the widely used NIST curve P-256 and more than 2 times faster than Curve25519 [4]. In comparison to other high-performance alternatives such as the genus 2 Kummer surface proposed by Gaudry and Schost [17], Four$\mathbb{Q}$ is, in most cases, more than 1.2x faster on x64 processors. For all of these comparisons, Costello and Longa's Four$\mathbb{Q}$ implementation (i) does not exploit vector instructions (in contrast to Curve25519 and Kummer implementations that do [7,11]), and (ii) is only optimized for x64 platforms. Therefore, the deployment and evaluation of Four$\mathbb{Q}$ on 32-bit ARM processors with NEON support, for which the use of vector instructions pose a different design paradigm, is still missing.

In this work, we engineer an efficient NEON-based implementation of Four$\mathbb{Q}$ targeting 32-bit ARM Cortex-A microarchitectures that are based on the widely used ARMv7 instruction set. Our design, although intended for high-performance applications, is not exclusive to only *one* microarchitecture; we analyze the different features from multiple Cortex-A microarchitectures and come up with an implementation that performs well across a wide range of ARM platforms. Specifically, our analysis includes *four* popular ARM processor cores: Cortex-A7, A8, A9 and A15. In addition, our implementation runs in constant-time, i.e., it is protected against timing and cache attacks [19], and supports the *three* core elliptic curve-based computations found in most cryptographic protocols (including Diffie-Hellman key exchange and digital signatures): variable-base, fixed-base and double-scalar multiplication. By considering these design decisions and functionality, we expect to ultimately produce practical software that can be used in real-world applications. Our code has been made publicly available as part of version 2.0 of Four$\mathbb{Q}$lib [13].

Our benchmark results extend Four$\mathbb{Q}$'s top performance to 32-bit ARM processors with NEON, and demonstrate for the first time Four$\mathbb{Q}$'s vectorization potential. For example, on a 2.0 GHz Odroid XU3 board powered by a Cortex-A15 CPU, our software computes a variable-base scalar multiplication in only 132,000 cycles (or 66 μs for a throughput above 15,150 operations/second). This result is about 1.7x faster than the Kummer implementation from [7], about 1.8x faster than the GLV+GLS based implementation from [15], about 2.4x faster than the Curve25519 implementation from [9], and about 5.6x faster than the implementation of the standardized NIST curve K-283 from [10]. As in our case, all of these implementations are state-of-the-art, exploit NEON instructions and are protected against timing and cache attacks. See Sect. 5 for complete benchmark results.

The paper is organized as follows. In Sect. 2, we provide relevant details about Fourℚ. In Sect. 3, we describe the 32-bit ARM architecture using NEON with focus on the targeted Cortex-A processors. We describe our vectorized NEON design and optimizations in Sect. 4 and, finally, Sect. 5 concludes the paper with the analysis and benchmark results.

2 The Fourℚ Curve

This section describes Fourℚ, where we adopt the notation from [12] for the most part. Fourℚ [12] is defined as the complete twisted Edwards [6] curve given by

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

where the quadratic extension field $\mathbb{F}_{p^2}=\mathbb{F}_p(i)$ for $i^2=-1$ and $p=2^{127}-1$, and $d=125317048443780598345676279555970305165\cdot i+4205857648805777768770$.

The $\mathbb{F}_{p^2}$-rational points lying on the curve Eq. (1) form an abelian group for which the neutral element is $\mathcal{O}_{\mathcal{E}}=(0,1)$ and the inverse of a point (x,y) is $(-x,y)$. The cardinality of this group is $\#\mathcal{E}(\mathbb{F}_{p^2})=392\cdot N$, where N is a 246-bit prime; thus, the prime-order subgroup $\mathcal{E}(\mathbb{F}_{p^2})[N]$ can be used to build cryptographic systems.

Fourℚ is equipped with *two* efficiently computable endomorphisms, ψ and ϕ, which give rise to a four-dimensional decomposition $m\mapsto(a_1,a_2,a_3,a_4)\in\mathbb{Z}^4$ for any integer $m\in[1,2^{256})$ such that $0\le a_i<2^{64}$ for $i=1,\ldots,4$ and such that a_1 is odd. This decomposition enables a four-dimensional variable-base scalar multiplication with the form

$$[m]P=[a_1]P+[a_2]\phi(P)+[a_3]\psi(P)+[a_4]\phi(\psi(P)),$$

for any point $P\in\mathcal{E}(\mathbb{F}_{p^2})[N]$.

The details of Fourℚ's variable-base scalar multiplication based on the four-dimensional decomposition are shown in Algorithm 1. The curve arithmetic is based on Hisil et al. explicit formulas that use *extended twisted Edwards coordinates* [18]: any projective tuple $(X:Y:Z:T)$ with $Z\neq0$ and $T=XY/Z$ corresponds to an affine point $(x,y)=(X/Z,Y/Z)$. Note that these formulas are also *complete* on $\mathcal{E}$, which means that they work without exceptions for all points in $\mathcal{E}(\mathbb{F}_{p^2})$.

The execution of Algorithm 1 begins with the computation of the endomorphisms ψ and ϕ, and the computation of the 8-point precomputed table (Steps 1−2). These precomputed points are stored in coordinates $(X+Y,Y-X,2Z,2dT)$ for efficiency. Scalar decomposition and multiscalar recoding are then applied to the input scalar m at Steps 3 and 4 as described in [12, Proposition 5] and [12, Algorithm 1], respectively. Finally, the main loop consists of 64 iterations each computing a point doubling (Step 7) and a point addition with a point from the precomputed table (Step 8). Following [12], the next coordinate representations are used throughout the algorithm: $\mathbf{R_1}:(X,Y,Z,T_a,T_b)$, such that $T=T_a\cdot T_b$, $\mathbf{R_2}:(X+Y,Y-X,2Z,2dT)$, $\mathbf{R_3}:(X+Y,Y-X,Z,T)$ and

Algorithm 1. FourQ's scalar multiplication on $\mathcal{E}(\mathbb{F}_{p^2})[N]$ (from [12]).

```
Input: Point $P \in \mathcal{E}(\mathbb{F}_{p^2})[N]$ and integer scalar $m \in [0, 2^{256})$.
Output: $[m]P$.
  Compute endomorphisms:
  1: Compute $\phi(P)$, $\psi(P)$ and $\psi(\phi(P))$.
  Precompute lookup table:
  2: Compute $T[u] = P + [u_0]\phi(P) + [u_1]\psi(P) + [u_2]\psi(\phi(P))$ for $u = (u_2, u_1, u_0)_2$ in $0 \leq u \leq 7$.
       Write $T[u]$ in coordinates $(X + Y, Y - X, 2Z, 2dT)$.
  Scalar decomposition and recoding:
  3: Decompose $m$ into the multiscalar $(a_1, a_2, a_3, a_4)$ as in [12, Proposition 5].
  4: Recode $(a_1, a_2, a_3, a_4)$ into $(d_{64}, \ldots, d_0)$ and $(m_{64}, \ldots, m_0)$ using [12, Algorithm 1].
       Write $s_i = 1$ if $m_i = -1$ and $s_i = -1$ if $m_i = 0$.
  Main loop:
  5: $Q = s_{64} \cdot T[d_{64}]$
  6: for $i = 63$ to $0$ do
  7:     $Q = [2]Q$
  8:     $Q = Q + s_i \cdot T[d_i]$
  9: return $Q$
```

$\mathbf{R_4}$: (X, Y, Z). In the main loop, point doublings are computed as $\mathbf{R_1} \leftarrow \mathbf{R_4}$ and point additions as $\mathbf{R_1} \leftarrow \mathbf{R_1} \times \mathbf{R_2}$, where the input using $\mathbf{R_1}$ comes from the output of a doubling (after ignoring coordinates T_a and T_b) and the input using $\mathbf{R_2}$ is a precomputed point from the table.

3 ARM NEON Architecture

The 32-bit RISC-based ARM architecture, which includes ARMv7, is the most popular architecture in mobile devices. It is equipped with 16 32-bit registers (`r0-r15`) and an instruction set supporting 32-bit operations or, in the case of Thumb and Thumb2, a mix of 16- and 32-bit operations. Many ARM cores include NEON, a powerful 128-bit SIMD engine that comes with 16 128-bit registers (`q0-q15`) which can also be viewed as 32 64-bit registers (`d0-d31`). NEON includes support for 8-, 16-, 32- and 64-bit signed and unsigned integer operations. For more information, refer to [2].

The following is a list of basic data processing instructions that are used in our NEON implementation. Since our design is based on a signed integer representation (see Sect. 4), most instructions below are specified with signed integer datatypes (denoted by `.sXX` in the instruction mnemonic). All of the timings provided correspond to Cortex-A8 and A9 (see [1,3]).

– `vmull.s32` performs 2 signed 32×32 multiplications and produces 2 64-bit products. When there are no pipeline stalls, the instruction takes 2 cycles.

- `vmlal.s32` performs 2 signed 32×32 multiplications, produces 2 64-bit products and accumulates the results with 2 64-bit additions. It has a cost similar to `vmull.s32`, which means that additions for accumulation are for free.
- `vadd.s64` and `vsub.s64` perform 1 or 2 signed 64-bit additions (resp. subtractions). When there are no pipeline stalls, the instruction takes 1 cycle.
- `vshl.i32` performs 2 or 4 32-bit logical shifts to the left by a fixed value. When there are no pipeline stalls, the instruction takes 1 cycle.
- `vshr.s64` and `vshr.u64` perform 1 or 2 64-bit arithmetic and logical (resp.) shifts to the right by a fixed value. It has a cost similar to `vshl.i32`.
- `vand.u64` performs a bitwise logical `and` operation. It has a cost similar to `vadd.s64`.
- `vbit` inserts each bit from the first operand into the destination operand if the corresponding bit in the second operand is 1. Otherwise, the destination bit remains unchanged. When there are no pipeline stalls, the instruction takes 1 cycle if operands are 64-bit long and 2 cycles if operands are 128-bit long.

Multiply and multiply-and-add instructions (`vmull.s32` and `vmlal.s32`) have latencies of 6 cycles. However, if a multiply-and-add follows a multiply or another multiply-and-add that depends on the result of the first instruction then a special forwarding enables the issuing of these instructions back-to-back. In this case, a series of multiply and multiply-and-add instructions can achieve a throughput of *two* cycles per instruction.

3.1 Targeted Platforms

Our implementation is optimized for the 32-bit Cortex-A series with ARMv7 support, with a special focus on Cortex-A7, A8, A9 and A15 cores. Next, we describe the most relevant architectural features that are considered in the design of our NEON-based software to achieve a consistent performance across microarchitectures.

Cortex-A7. This microarchitecture has in-order execution, partial dual-issue and a NEON engine capable of executing (at most) one NEON arithmetic operation per cycle.

Cortex-A8. This microarchitecture has the NEON pipeline logically behind the integer pipeline. Once NEON instructions flow through the integer pipeline, they are stored in a queue getting ready for execution. This queue accumulates instructions faster than what it takes to execute them, which means that the integer pipeline can execute ARM instructions in the background while the NEON unit is busy. This is exploited in Sect. 4.2 to boost the performance of the $\mathbb{F}_{p^2}$ implementation by mixing ARM and NEON instructions. Cortex-A8 also has highly flexible dual-issue capabilities that support many combinations of ARM and NEON instructions; for example, Cortex-A8 can issue a NEON load/store instruction with a NEON arithmetic instruction in one cycle. The NEON engine has one execution port for arithmetic instructions and one execution port for load/store/permute instructions; this enables back-to-back execution of pairs of NEON {load/store, arithmetic} instructions (see Sect. 4.2).

Cortex-A9. This microarchitecture no longer has the NEON unit (with a detached NEON queue) behind the integer unit as in Cortex-A8. This significantly reduces the cost of NEON to ARM data transfers, but also reduces the efficiency gain obtained by mixing ARM and NEON instructions. In addition, NEON on Cortex-A9 has some limitations: load/store instructions have longer latency, and there is no dual-issue support. To minimize the inefficiency of the load/store port it is possible to interleave these instructions with other instructions, as detailed in Sect. 4.2.

Cortex-A15. This microarchitecture has out-of-order execution on both ARM and NEON units. The NEON engine, which is fully integrated with the ARM core, is capable of executing two operations per cycle. The ARM and NEON load/store ports are also integrated. In many platforms, Cortex-A15 cores are paired with Cortex-A7 cores to form powerful heterogeneous systems (a.k.a. big.LITTLE).

4 Vectorization Using NEON

The basic feature to consider in a NEON-based design is that vector multiplication is capable of working over *two pairs* of 32-bit values to produce *one pair* of 64-bit products.

In our preliminary analysis, we considered two approaches for vectorization:

– Vectorization across different $\mathbb{F}_{p^2}$ multiplications and squarings inside point formulas.
– Vectorization across different field multiplications inside $\mathbb{F}_{p^2}$ multiplications and squarings.

The first option has the disadvantage that pairing of $\mathbb{F}_{p^2}$ operations inside point addition and doubling formulas is not perfect and would lead to sub-optimal performance. E.g., the 3 squarings in the doubling formula would be computed either as 2 pairs of squarings (increasing the cost in 1 squaring) or as 1 pair of squarings and 1 pair of multiplications, using any available multiplication (degrading the speed of 1 squaring). This approach also puts extra pressure on register allocation, which can potentially lead to a high number of memory accesses. In contrast, the second approach can benefit from the availability of independent operations over $\mathbb{F}_p$ inside the $\mathbb{F}_{p^2}$ arithmetic. Both multiplications and squarings over $\mathbb{F}_{p^2}$ naturally contain *pairs* of field multiplications; all multiplications are independent from each other and, therefore, can be optimally paired for NEON vector multiplication.

We chose the second vectorization option for our implementation, which is described next.

4.1 Vectorization of $\mathbb{F}_{(2^{127}-1)^2}$ Arithmetic

For our design we use radix $t = 2^{26}$ and represent a quadratic extension field element $c = a + b \cdot i \in \mathbb{F}_{p^2}$ using $a = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4$ and $b = b_0 + b_1 t + b_2 t^2 + b_3 t^3 + b_4 t^4$. In a similar fashion to Naehrig et al.'s interleaving strategy [21], in our implementation the *ten-coefficient vector* representing element c is stored "interleaved" as $(b_4, a_4, b_3, a_3, b_2, a_2, b_1, a_1, b_0, a_0)$ in little endian format, i.e., a_0 and b_4 are stored in the lowest and highest memory addresses, respectively. Each coefficient is *signed* and occupies 32 bits in memory; however, when fully reduced, coefficients $a_0, b_0, \dots, a_3, b_3$ have values in the range $[0, 2^{26})$ and coefficients a_4 and b_4 have values in the range $[0, 2^{23})$.

Using the representation above, addition and subtraction of two elements in $\mathbb{F}_{p^2}$ are simply done with 2 128-bit vector addition instructions (resp. subtractions) and 1 64-bit vector addition instruction (resp. subtraction) using the NEON instruction `vadd.s32` (resp. `vsub.s32`). The corresponding results are immediately produced in the interleaved representation.

For the case of multiplication and squaring, we base the implementation on a schoolbook-like multiplication that includes the reduction modulo $p = 2^{127} - 1$. Given two field elements $a = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4$ and $b = b_0 + b_1 t + b_2 t^2 + b_3 t^3 + b_4 t^4$, multiplication modulo $(2^{127} - 1)$ can be computed by

$$\begin{aligned}
c_0 &= a_0 b_0 + 8(a_1 b_4 + a_4 b_1 + a_2 b_3 + a_3 b_2)\\
c_1 &= a_0 b_1 + a_1 b_0 + 8(a_2 b_4 + a_4 b_2 + a_3 b_3)\\
c_2 &= a_0 b_2 + a_2 b_0 + a_1 b_1 + 8(a_3 b_4 + a_4 b_3)\\
c_3 &= a_0 b_3 + a_3 b_0 + a_1 b_2 + a_2 b_1 + 8(a_4 b_4)\\
c_4 &= a_0 b_4 + a_4 b_0 + a_1 b_3 + a_3 b_1 + a_2 b_2.
\end{aligned} \tag{2}$$

Next, we show how to use (2) in the vectorized computation of multiplication and squaring over $\mathbb{F}_{p^2}$. Note that the operation sequences below are designed to maximize performance and to fit all intermediate computations in the 16 128-bit NEON registers at our disposal.

Multiplication in $\mathbb{F}_{p^2}$. Let $A = (b_4, a_4, b_3, a_3, b_2, a_2, b_1, a_1, b_0, a_0)$ and $B = (d_4, c_4, d_3, c_3, d_2, c_2, d_1, c_1, d_0, c_0)$ be coefficient vectors that represent elements $(a + b \cdot i) \in \mathbb{F}_{p^2}$ and $(c + d \cdot i) \in \mathbb{F}_{p^2}$, respectively. To multiply these two elements, we first shift A to the left by 3 bits to obtain

$$t_1 = (8b_4, 8a_4, \dots, 8b_1, 8a_1, 8b_0, 8a_0),$$

which requires 1 64-bit and 2 128-bit vector shifts using `vshl.i32`.

We then compute the first *three* terms of the multiplications $bc = b \times c$ and $bd = b \times d$, by multiplying in pairs $(b_0 d_0, b_0 c_0)$, $(8b_1 d_4, 8b_1 c_4)$, $(8b_4 d_1, 8b_4 c_1)$ and so on, and accumulating the intermediate values to produce

$$\begin{aligned}
(bd_0, bc_0) &= (b_0 d_0 + 8b_1 d_4 + \dots + 8b_3 d_2, b_0 c_0 + 8b_1 c_4 + \dots + 8b_3 c_2)\\
(bd_1, bc_1) &= (b_0 d_1 + b_1 d_0 + \dots + 8b_3 d_3, b_0 c_1 + b_1 c_0 + \dots + 8b_3 c_3)\\
(bd_2, bc_2) &= (b_0 d_2 + b_2 d_0 + \dots + 8b_4 d_3, b_0 c_2 + b_2 c_0 + \dots + 8b_4 c_3).
\end{aligned}$$

The computation above is executed using (2). In total (including the missing two terms that are computed later on), it requires 25 vector multiplications: 5 are computed using `vmull.s32` and 20 are computed using `vmlal.s32`. Additions are not counted because they are virtually absorbed by the multiply-and-add instructions.

Then, we compute the *five* terms of the multiplications $ac = a \times c$ and $ad = a \times d$. Similarly to above, we compute pairwise multiplications (a_0d_0, a_0c_0), $(8a_1d_4, 8a_1c_4)$, $(8a_4d_1, 8a_4c_1)$ and so on, and accumulate the intermediate values to produce

$$\begin{aligned}
(ad_0, ac_0) &= (a_0d_0 + 8a_1d_4 + \ldots + 8a_3d_2, a_0c_0 + 8a_1c_4 + \ldots + 8a_3c_2)\\
(ad_1, ac_1) &= (a_0d_1 + a_1d_0 + \ldots + 8a_3d_3, a_0c_1 + a_1c_0 + \ldots + 8a_3c_3)\\
(ad_2, ac_2) &= (a_0d_2 + a_2d_0 + \ldots + 8a_4d_3, a_0c_2 + a_2c_0 + \ldots + 8a_4c_3)\\
(ad_3, ac_3) &= (a_0d_3 + a_3d_0 + \ldots + 8a_4d_4, a_0c_3 + a_3c_0 + \ldots + 8a_4c_4)\\
(ad_4, ac_4) &= (a_0d_4 + a_4d_0 + \ldots + a_2d_2, a_0c_4 + a_4c_0 + \ldots + a_2c_2).
\end{aligned}$$

As before, this vectorized schoolbook computation requires 25 multiplications: 5 computed using `vmull.s32` and 20 computed using `vmlal.s32`.

The intermediate values computed so far are subtracted and added to obtain the first *three* terms of the results $r = ac - bd$ and $s = ad + bc$. This requires 3 64-bit vector additions using `vadd.s64` and 3 64-bit vector subtractions using `vsub.s64`:

$$\begin{aligned}
(s_0, r_0) &= (ad_0 + bc_0, ac_0 - bd_0)\\
(s_1, r_1) &= (ad_1 + bc_1, ac_1 - bd_1)\\
(s_2, r_2) &= (ad_2 + bc_2, ac_2 - bd_2).
\end{aligned}$$

We then compute the remaining *two* terms in the computation of $bc = b \times c$ and $bd = b \times d$ (i.e., (bd_3, bc_3) and (bd_4, bc_4)) as follows

$$\begin{aligned}
(bd_3, bc_3) &= (b_0d_3 + b_3d_0 + \ldots + 8b_4d_4, b_0c_3 + b_3c_0 + \ldots + 8b_4c_4)\\
(bd_4, bc_4) &= (b_0d_4 + b_4d_0 + \ldots + b_2d_2, b_0c_4 + b_4c_0 + \ldots + b_2c_2).
\end{aligned}$$

Finally, we complete the computation with the last *two* terms of the results $r = ac - bd$ and $s = ad + bc$. This involves 2 64-bit vector additions using `vadd.s64` and 2 64-bit vector subtractions using `vsub.s64`:

$$\begin{aligned}
(s_3, r_3) &= (ad_3 + bc_3, ac_3 - bd_3)\\
(s_4, r_4) &= (ad_4 + bc_4, ac_4 - bd_4).
\end{aligned}$$

The coefficients in the resulting vector $(s_4, r_4, \ldots, s_0, r_0)$ need to be reduced before they are used by subsequent multiplications or squarings. We explain this process below, after discussing squarings in $\mathbb{F}_{p^2}$.

Squaring in $\mathbb{F}_{p^2}$. Let $A = (b_4, a_4, b_3, a_3, b_2, a_2, b_1, a_1, b_0, a_0)$ be a coefficient vector representing an element $(a + b \cdot i)$ in $\mathbb{F}_{p^2}$. To compute the squaring of this element we first shift its coefficients to the right to obtain

$$t_1 = (0, b_4, 0, b_3, 0, b_2, 0, b_1, 0, b_0),$$

which requires 1 64-bit and 2 128-bit vector shifts using `vshr.u64`.

Then, A is subtracted and added with t_1 to obtain

$$t_2 = (b_4, a_4 - b_4, b_3, a_3 - b_3, b_2, a_2 - b_2, b_1, a_1 - b_1, b_0, a_0 - b_0)$$
$$t_3 = (b_4, a_4 + b_4, b_3, a_3 + b_3, b_2, a_2 + b_2, b_1, a_1 + b_1, b_0, a_0 + b_0),$$

which requires 1 64-bit and 2 128-bit vector additions using `vadd.s32` and 1 64-bit and 2 128-bit vector subtractions using `vsub.s32`.

We then shift A to the left by one bit with 1 64-bit and 2 128-bit vector shifts using `vshl.i32`, as follows

$$t_4 = (2a_4, 0, 2a_3, 0, 2a_2, 0, 2a_1, 0, 2a_0, 0).$$

We perform a bitwise selection over t_2 and t_4 using 3 `vbit` instructions to obtain

$$t_5 = (2a_4, a_4 - b_4, 2a_3, a_3 - b_3, 2a_2, a_2 - b_2, 2a_1, a_1 - b_1, 2a_0, a_0 - b_0).$$

We then shift the result by 3 bits to the left using 1 64-bit and 2 128-bit vector shifts with `vshr.u64`, as follows

$$t_6 = (16a_4, 8(a_4 - b_4), 16a_3, 8(a_3 - b_3), 16a_2, 8(a_2 - b_2), 16a_1, 8(a_1 - b_1), 16a_0, 8(a_0 - b_0)).$$

We then compute the *five* terms of the multiplications $r = (a + b) \times (a - b)$ and $s = 2a \times b$. As before, we compute pairwise multiplications $(2a_0b_0, (a_0 - b_0)(a_0 + b_0))$, $(16a_1b_4, 8(a_1 - b_1)(a_4 + b_4))$, $(16a_4b_1, 8(a_4 - b_4)(a_1 + b_1))$ and so on, and accumulate the intermediate values to produce

$$(s_0, r_0) = (2a_0b_0 + \ldots + 16a_3b_2, (a_0 - b_0)(a_0 + b_0) + \ldots + 8(a_3 - b_3)(a_2 + b_2))$$
$$(s_1, r_1) = (2a_0b_1 + \ldots + 16a_3b_3, (a_0 - b_0)(a_1 + b_1) + \ldots + 8(a_3 - b_3)(a_3 + b_3))$$
$$(s_2, r_2) = (2a_0b_2 + \ldots + 16a_4b_3, (a_0 - b_0)(a_2 + b_2) + \ldots + 8(a_4 - b_4)(a_3 + b_3))$$
$$(s_3, r_3) = (2a_0b_3 + \ldots + 16a_4b_4, (a_0 - b_0)(a_3 + b_3) + \ldots + 8(a_4 - b_4)(a_4 + b_4))$$
$$(s_4, r_4) = (2a_0b_4 + \ldots + 2a_2b_2, (a_0 - b_0)(a_4 + b_4) + \ldots + (a_2 - b_2)(a_2 + b_2)).$$

As before, this computation follows (2) and involves 5 multiplications using `vmull.s32` and 20 multiplications using `vmlal.s32`. The reduction procedure that needs to be applied to the output vector $(s_4, r_4, \ldots, s_0, r_0)$ before subsequent multiplications or squarings is described next.

Coefficient Reduction. After computing a multiplication or squaring over $\mathbb{F}_{p^2}$, resulting coefficients must be reduced to avoid overflows in subsequent operations. Given a coefficient vector $(s_4, r_4, \ldots, s_0, r_0)$, coefficient reduction can be accomplished by applying a chain of shifts, additions and logical `and` instructions using the flow $(s_0, r_0) \rightarrow (s_1, r_1) \rightarrow (s_2, r_2) \rightarrow (s_3, r_3) \rightarrow (s_4, r_4) \rightarrow (s_0, r_0)$. In total, this requires 7 vector shifts using `vshr.s64`, 6 vector `and` operations using `vand.u64`, and 6 vector additions using `vadd.s64`. This chain of operations, however, introduces many data hazards that can stall the pipeline for several cycles. In our implementation, for computations in which instruction rescheduling is unable to eliminate most of these data hazards, we switch to a different alternative that consists of splitting the computation in the following *two* propagation chains: $(s_0, r_0) \rightarrow (s_1, r_1) \rightarrow (s_2, r_2) \rightarrow (s_3, r_3) \rightarrow (s_4, r_4)$, and $(s_3, r_3) \rightarrow (s_4, r_4) \rightarrow (s_0, r_0)$. Even though this approach increases the operation count in 1 vector shift, 1 vector addition and 1 vector `and`, it allows to speed up the overall computation because both chains can be interleaved, which eliminates all data hazards.

Vector-Instruction Count. Based on the operation description above, multiplication over $\mathbb{F}_{p^2}$ involves 11 shifts, 7 logical `and` instructions, 17 additions and 50 multiplications. Similarly, squaring over $\mathbb{F}_{p^2}$ involves 17 shifts, 7 logical `and` instructions, 3 bit-selection instructions, 13 additions and 25 multiplications. These counts include coefficient reduction.

4.2 Additional Optimizations to the $\mathbb{F}_{p^2}$ Implementation

As explained in Sect. 3.1, the ARM architecture with NEON support opens the possibility of optimizing software by exploiting the instruction-level parallelism between ARM and NEON instruction sets. We remark, however, that the capability of boosting performance by exploiting this feature strongly depends on the specifics of the targeted microarchitecture and application. For example, microarchitectures such as Cortex-A8 have a relatively large NEON instruction queue that keeps the NEON execution units busy once it is filled; when this happens the ARM core can execute ARM instructions virtually *in parallel*. In contrast, other microarchitectures such as Cortex-A7 and Cortex-A15 exhibit a more continuous flow of instructions to the NEON execution ports, which means that gaining efficiency from mixing ARM and NEON instructions gets significantly more challenging. This is especially true for implementations that rely on the full power of vector instructions. We note, however, that the technique could still be beneficial for implementations that generate enough pipeline stalls. In this case, NEON pipeline stalls could give enough room for ARM instructions to run while the NEON engine recovers (e.g., see [15]).

In the case of our NEON implementation, careful scheduling of instructions was effective in dealing with most latency problems inside the $\mathbb{F}_{p^2}$ functions and, thus, we were able to minimize the occurrence of pipeline stalls. We verified experimentally that this makes very difficult to obtain any additional speedup by mixing ARM and NEON instructions on microarchitectures such as Cortex-A7 and A15. In the case of microarchitectures that are more favorable to the instruction mixing technique (e.g., Cortex-A8 and A9), we applied the following approach. We use NEON to perform the relatively expensive multiplications and squarings over $\mathbb{F}_{p^2}$, and ARM to execute the simpler additions and subtractions (or any combination of these operations). To do this, we inserted add/sub ARM code into the larger NEON-based functions, carefully interleaving NEON and ARM instructions.

We verified that instantiating NEON-based multiplications and squarings that include ARM-based additions or subtractions effectively reduces the cost of these smaller operations. We do even better by suitably merging additions and subtractions inside NEON functions. Specifically, we have identified and implemented the following combinations of operations over $\mathbb{F}_{p^2}$ after analyzing twisted Edwards point doubling and addition formulas:

- `MulAdd`: multiplication $a \times b$ using NEON, addition $c + d$ using ARM.
- `MulSub`: multiplication $a \times b$ using NEON, subtraction $c - d$ using ARM.
- `MulDblSub`: multiplication $a \times b$ using NEON, doubling/subtraction $2 \times c - d$ using ARM.
- `MulAddSub`: multiplication $a \times b$ using NEON, addition $c + d$ and subtraction $c - d$ using ARM.
- `SqrAdd`: squaring a^2 using NEON, addition $c + d$ using ARM.
- `SqrAddSub`: squaring a^2 using NEON, addition $c + d$ and subtraction $c - d$ using ARM.

In our software, the use of these functions is optional. Users can enable this optimization by setting a command flag called "MIX_ARM_NEON". Following the details above, we suggest turning this flag on for Cortex-A8 and A9, and turning it off for Cortex-A7 and A15. See Appendix A for details about the use of these functions inside point doubling and addition.

Additionally, we improve the performance of multiplication and squaring over $\mathbb{F}_{p^2}$ even further by interleaving load/store operations with arithmetic operations. As explained in Sect. 3.1, microarchitectures such as Cortex-A8 are capable of executing one load or store instruction and one arithmetic instruction back-to-back. On the other hand, Cortex-A9 load/store instructions suffer from longer latencies. It is quite fortunate that, in both cases, suitable interleaving of load/store instructions with other non-memory instructions does benefit performance (albeit under different circumstances). We observed experimentally that some newer processors such as Cortex-A15 are negatively impacted by such interleaving. Since in our code input loading and output storing only occur at the very top and at the very bottom of $\mathbb{F}_{p^2}$ arithmetic functions, resp., it was straightforward to create two different execution paths with minimal impact to code size. The path selection is done at compile time: users can enable the optimization by setting a command flag called "INTERLEAVE". We suggest turning this flag on for Cortex-A7, A8 and A9, and turning it off for Cortex-A15.

4.3 Putting Pieces Together

We now describe the implementation details of other necessary operations, and explain how these and our vectorized functions are assembled together to compute Algorithm 1.

Since our vectorization approach is applied at the $\mathbb{F}_{p^2}$ level, most high-level functions in the scalar multiplication remain unchanged for the most part (relative to a non-vectorized implementation). Hence, in our software, endomorphism and point formulas, which are used for table precomputation and in the main loop of Algorithm 1 (Steps 1−2, 7−8), are implemented with only a few minor modifications in comparison with the original explicit formulas. Refer to Appendix A for the modified doubling and addition formulas used in our implementation.

The functions for scalar decomposition and recoding (Steps 3−4) are directly implemented as detailed in [12, Proposition 5] and [12, Algorithm 1], respectively. To extract points from the precomputed table, which is required at Step 8, we carry out a linear pass over the full content of the table performing bitwise selections with `vbit` instructions. At each step, a mask computed in constant-time determines if a given value is "taken" or not. Inversion over $\mathbb{F}_{p^2}$, which is required for final conversion to affine coordinates at the very end of Algorithm 1, involves the computation of field multiplications and squarings. For these operations, we represent a field element a as a coefficient vector $(a_4, a_3, a_2, a_1, a_0)$, and apply the schoolbook computation (2) (exploiting the typical savings for the case of squarings). In this case, vector multiplications are applied over pairs of internal integer multiplications. This pairing is not optimal, but the effect over the overall cost is relatively small.

Finally, we implemented straightforward functions to convert back and forth between our $\mathbb{F}_{p^2}$ vector representation and the canonical representation. These functions are required just once at the beginning of scalar multiplication to convert the input point to vector representation, and once at the very end to convert the output point to canonical representation. In addition, we need to perform one conversion from $\mathbb{F}_{p^2}$ to $\mathbb{F}_p$ vector representation (and one conversion back) when computing a modular inversion during the final conversion to affine coordinates.

In order to protect against timing and cache attacks, our implementation does not contain branches that depend on secret data and does not use secret memory addresses. For the most part, the elimination of secret branches is greatly facilitated by the regular structure of Four$\mathbb{Q}$'s algorithms [12], whereas the elimination of secret memory addresses is done by performing linear passes over the full content of tables in combination with some masking technique.

5 Implementation and Results

In this section, we carry out a theoretical analysis on the core scalar multiplication operations and then present benchmark results on a large variety of ARM Cortex-A based platforms: a 0.9 GHz Raspberry Pi 2 with a Cortex-A7 processor, a 1.0 GHz BeagleBone board with a Cortex-A8 processor, a 1.7 GHz Odroid X2 with a Cortex-A9 processor and a 2.0 GHz Odroid XU3 with a Cortex-A15 processor. All of these ARM-based devices come equipped with a NEON vector unit. The software was compiled with GNU GCC v4.7.2 for the case of

Raspberry Pi and BeagleBone, and with GNU GCC v4.8.2 for the case of the Odroid devices. We report the average of 10^4 operations which were measured with the `clock_gettime()` function and scaled to clock cycles using the processor frequency.

Next, we analyze the different scalar multiplications when using Fourℚ.

Variable-Base Scalar Multiplication. Following Algorithm 1, this operation involves the computation of 1 ϕ endomorphism, 2 ψ endomorphisms and 7 points additions in the precomputation stage; 64 doublings, 64 additions and 65 constant-time 8-point table lookups (denoted by **lut8**) in the evaluation stage; and, finally, 1 inversion and 2 multiplications over $\mathbb{F}_{p^2}$ for converting the final result to affine coordinates. This represents a cost of 1**i** + 842**m** + 283**s** + 950.5**a** + 65**lut8** or 3948**M** + 128**S** + 4436**A** + 65**lut8** (considering that 1**m** = 4**M** + 2**A** using schoolbook multiplication and that 1**s** = 2**M** + 3**A**[1]). This operation count does not include other relatively small computations, such as decomposition and recoding. We consider that field inversion of an element a is computed as $a^{2^{127}-3} \bmod (2^{127}-1)$ using a fixed chain consisting of 12 modular multiplications and 126 modular squarings.

Fixed-Base Scalar Multiplication. We implemented this operation using the mLSB-set comb method proposed by Faz-Hernández, Longa and Sánchez (see [14, Algorithm 5]). Recall that scalars are in the range $[0, 2^{256})$. By applying a relatively inexpensive Montgomery reduction, a given input scalar can be reduced to the range $[0, N)$ and, thus, fix the maximum scalar bitlength to $t = 249$. As an example, consider the table parameters $w = 5$ and $v = 5$. In this case, the mLSB-set comb method costs $\lceil \frac{249}{w \cdot v} \rceil - 1 = 9$ doublings and $v \lceil \frac{249}{w \cdot v} \rceil - 1 = 49$ mixed additions using $v \cdot 2^{w-1} = 80$ points computed *offline*. Since precomputed points are stored in coordinates $(x + y, y - x, 2t)$ the storage requirement is 7.5 KB and the operation cost is roughly given by 1**i** + 372**m** + 36**s** + 397**a** + 49**lut16** or 1574**M** + 128**S** + 1648**A** + 49**lut16**. This estimate does not include the cost of scalar recoding and conversion.

Double-Scalar Multiplication. We implemented this operation using width-w non-adjacent form (wNAF) with interleaving [16]. Given a computation with the form $[k]P + [l]Q$, scalars k and l can be split in *four* 64-bits sub-scalars each using Fourℚ's decomposition algorithm. After converting the eight sub-scalars to w-NAF, we compute an 8-way multiscalar multiplication as the main operation. As an example, consider window parameters $w_P = 8$ and $w_Q = 4$ (this assumes that the point P is known in advance, which typically happens in signature verification algorithms). In this case, the computation involves 4

[1] **I**, **M**, **S** and **A** represent the cost of modular inversion, multiplication, squaring and addition using $p = 2^{127} - 1$ (resp.); **i**, **m**, **s** and **a** represent the cost of inversion, multiplication, squaring and addition over $\mathbb{F}_{(2^{127}-1)^2}$ (resp.).

doublings and $4 \cdot (2^{w_Q-2} - 1) = 12$ additions (for the *online* precomputation), $4 \cdot (\frac{64}{w_P+1}) = 32$ mixed additions, $4 \cdot (\frac{64}{w_Q+1}) - 1 = 52$ additions and 63 doublings (for the evaluation stage) using $4 \cdot 2^{w_P-2} = 256$ points computed *offline*. Again, we store points in coordinates $(x+y, y-x, 2t)$. This fixes the storage requirement to 24 KB; the operation cost is roughly $1\mathbf{i} + 951\mathbf{m} + 268\mathbf{s} + 1034\mathbf{a}$ or $4354\mathbf{M} + 128\mathbf{S} + 4776\mathbf{A}$. This estimate does not include the cost of 2 scalar decompositions and 8 recordings to wNAF. E.g., assuming that $1\mathbf{S} = 0.8\mathbf{M}$ and $1\mathbf{A} = 0.1\mathbf{M}$, double-scalar multiplication is expected to be roughly 10% more expensive than variable-base on Four$\mathbb{Q}$.

5.1 Results

Table 1 includes benchmark results of our vectorized Four$\mathbb{Q}$ implementation for computing all of the core scalar multiplication operations. The results highlight the efficiency gain that can be obtained through the use of fixed-base scalar multiplications (e.g., during signature generation or ephemeral Diffie-Hellman key generation) using a relatively small amount of precomputation. Most notably, these results show for the first time the potential of using Four$\mathbb{Q}$ for signature verification: one double-scalar multiplication is, in most cases, less than 15% more expensive than single variable-base scalar multiplication.

Table 1. Performance results (in terms of thousands of cycles) of core scalar multiplication operations on Four$\mathbb{Q}$ with protection against timing and cache attacks on various ARM Cortex-A processors with NEON support. Results were rounded to the nearest 10^3 clock cycles. For this benchmark, fixed-base scalar multiplication used a precomputed table of 80 points (7.5 KB of storage) and double-scalar multiplication used a precomputed table of 256 points (24 KB of storage).

Scalar multiplication	Cortex-A7	Cortex-A8	Cortex-A9	Cortex-A15
$[k]P$, variable base	373	235	256	132
$[k]P$, fixed base	204	144	145	84
$[k]P + [l]Q$	431	269	290	155

In Table 2, we compare our results for variable-base scalar multiplication with other NEON-based implementations in the literature. We include results for the twisted Edwards GLV+GLS curve defined over $\mathbb{F}_{(2^{127}-5997)^2}$ that was proposed by Longa and Sica [20] and the genus 2 Kummer surface defined over $\mathbb{F}_{2^{127}-1}$ that was proposed by Gaudry and Schost [17]. These two curves, which we refer to as "GLV+GLS" and "Kummer", were the previous speed-record holders before the advent of Four$\mathbb{Q}$. Our comparisons also include the popular Montgomery curve known as "Curve25519", which is defined over $\mathbb{F}_{2^{255}-19}$ [4], and two binary curve

Table 2. Performance results (expressed in terms of thousands of clock cycles) of state-of-the-art implementations of various curves targeting the 128-bit security level for computing variable-base scalar multiplication on various ARM Cortex-A processors with NEON support. Results were rounded to the nearest 10^3 clock cycles. The benchmarks for Fourℚ were done on a 0.9 GHz Raspberry Pi 2 (Cortex-A8), a 1.0 GHz BeagleBone (Cortex-A8), a 1.7 GHz Odroid X2 (Cortex-A9) and a 2.0 GHz Odroid XU3 (Cortex-A15). Cortex-A8 and A9 benchmarks for the Kummer implementation [7] and Cortex-A8, A9 and A15 benchmarks for the Curve25519 implementation [9] were taken from eBACS [8] (computers "h7beagle", "h7green" and "sachr"), while Cortex-A7 benchmarks for Kummer and Curve25519 and Cortex-A15 benchmarks for Kummer were obtained by running eBACS' SUPERCOP toolkit on the corresponding targeted platform. The benchmarks for the GLV-GLS curve were taken directly from [15], and the benchmarks for the binary Koblitz curve K-283 and the binary Edwards curve B-251 were taken directly from [10].

Work	Curve	Cortex A7	Cortex A8	Cortex A9	Cortex A15
This work	Fourℚ	**373**	**235**	**256**	**132**
Bernstein et al. [7]	Kummer	580	305	356	224
Faz-Hernández et al. [15]	GLV+GLS	-	-	417	244
Bernstein et al. [9]	Curve25519	926	497	568	315
Câmara et al. [10]	B-251	-	657	789	511
Câmara et al. [10]	K-283	-	934	1,148	736

alternatives: the binary Edwards curve defined over $\mathbb{F}_{2^{251}}$ [5], referred to as "B-251", and the NIST's standard Koblitz curve K-283 [23], which is defined over the binary field $\mathbb{F}_{2^{283}}$.

Using the operation counts above and those listed in [12, Table 2], one can determine that Fourℚ's variable-base scalar multiplication is expected to be roughly 1.28 times faster than Kummer's ladder computation (assuming that 1**I** = 115**M**, 1**S** = 0.8**M**, 1**A** = 0.1**M** and 1 word-mul = 0.25**M**). Our actual results show that Fourℚ is between 1.3 and 1.7 times faster than Bernstein et al.'s Kummer implementation [7] on different ARM microarchitectures. Therefore, Fourℚ performs even better than expected, demonstrating that its efficient and compact arithmetic enable vector-friendly implementations. These results also highlight the effectiveness of the techniques described in Sect. 4.2.

In comparison to Curve25519, our NEON implementation is between 2.1 and 2.5 times faster when computing variable-base scalar multiplication. Our implementation is also significantly faster than state-of-the-art NEON implementations using binary curves; e.g., it is between 4 and 5.6 times faster than the implementation based on the NIST's standard K-283 curve.

In some cases, even larger speedups are observed for scenarios in which one can exploit precomputations. For example, for signature signing one can leverage the efficiency of fixed-base scalar multiplications to achieve between factor-2.1 and factor-2.8 speedups in comparison to the Kummer surface from [17], which does not support these efficient operations that exploit precomputations.

Acknowledgments. We thank Craig Costello for his valuable comments.

A Algorithms for Point Operations

The basic point doubling and addition functions used in the NEON implementation are shown in Algorithms 2 and 3, respectively. When selector "MIX_ARM_NEON" is enabled, the algorithms use the functions with the labels on the right (`MulAddSub`, `SqrAdd`, etc.), which mix ARM and NEON instructions as described in Sect. 4.2.

Algorithm 2. Point doubling using homogeneous/extended homogeneous coordinates on $\mathcal{E}$.

Input: $P = (X_1, Y_1, Z_1) \in \mathcal{E}(\mathbb{F}_{p^2})$.
Output: $2P = (X_2, Y_2, Z_2, T_{2,a}, T_{2,b}) \in \mathcal{E}(\mathbb{F}_{p^2})$.

```
 1: if MIX_ARM_NEON = true then
 2:     $t_1 = X_1^2,\ X_2 = X_1 + Y_1$                                  {SqrAdd}
 3:     $t_2 = Y_1^2$
 4:     $Z_2 = Z_1^2,\ T_{2,b} = t_1 + t_2,\ t_1 = t_2 - t_1$             {SqrAddSub}
 5:     $T_{2,a} = X_2^2$
 6:     $Y_2 = t_1 \times T_{2,b},\ t_2 = 2Z_2 - t_1$                    {MulDblSub}
 7:     $Z_2 = t_1 \times t_2,\ T_{2,a} = T_{2,a} - T_{2,b}$              {MulSub}
 8:     $X_2 = t_2 \times T_{2,a}$
 9: else
10:     $t_1 = X_1^2$
11:     $t_2 = Y_1^2$
12:     $X_2 = X_1 + Y_1$
13:     $T_{2,b} = t_1 + t_2$
14:     $t_1 = t_2 - t_1$
15:     $t_2 = Z_1^2$
16:     $T_{2,a} = X_2^2$
17:     $t_2 = t_2 + t_2$
18:     $t_2 = t_2 - t_1$
19:     $T_{2,a} = T_{2,a} - T_{2,b}$
20:     $Y_2 = t_1 \times T_{2,b}$
21:     $X_2 = t_2 \times T_{2,a}$
22:     $Z_2 = t_1 \times t_2$
23: endif
24: return $2P = (X_2, Y_2, Z_2, T_{2,a}, T_{2,b})$.
```

Algorithm 3. Point addition using extended homogeneous coordinates on $\mathcal{E}$.

Input: $P, Q \in \mathcal{E}(\mathbb{F}_{p^2})$ such that $P = (X_1, Y_1, Z_1, T_{1,a}, T_{1,b})$ and $Q = (X_2 + Y_2, Y_2 - X_2, 2Z_2, 2dT_2)$.
Output: $P + Q = (X_3, Y_3, Z_3, T_{3,a}, T_{3,b}) \in \mathcal{E}(\mathbb{F}_{p^2})$.

```
1: if MIX_ARM_NEON = true then
2:     $T_{3,a} = T_{1,a} \times T_{1,b},\ T_{3,b} = X_1 + Y_1,\ Y_3 = Y_1 - X_1$        {MulAddSub}
3:     $t_1 = 2Z_2 \times Z_1$
4:     $Z_3 = (2dT_2) \times T_{3,a}$
5:     $X_3 = (X_2 + Y_2) \times T_{3,b},\ t_2 = t_1 - Z_3,\ t_1 = t_1 + Z_3$        {MulAddSub}
6:     $Y_3 = (Y_2 - X_2) \times Y_3$
7:     $Z_3 = t_1 \times t_2,\ T_{3,a} = X_3 + Y_3,\ T_{3,b} = X_3 - Y_3$
8:     $X_3 = T_{3,b} \times t_2$
9:     $Y_3 = T_{3,a} \times t_1$
10: else
11:     $t_1 = X_1 + Y_1$
12:     $t_2 = Y_1 - X_1$
13:     $t_3 = T_{1,a} \times T_{1,b}$
14:     $t_4 = 2Z_2 \times Z_1$
15:     $Z_3 = (2dT_2) \times t_3$
16:     $X_3 = (X_2 + Y_2) \times t_1$
17:     $Y_3 = (Y_2 - X_2) \times t_2$
18:     $T_{3,a} = X_3 + Y_3$
19:     $T_{3,b} = X_3 - Y_3$
20:     $t_3 = t_1 - Z_3$
21:     $t_1 = t_1 + Z_3$
22:     $X_3 = T_{3,b} \times t_3$
23:     $Z_3 = t_3 \times t_4$
24:     $Y_3 = T_{3,a} \times t_4$
25: endif
26: return $P + Q = (X_3, Y_3, Z_3, T_{3,a}, T_{3,b})$.
```

References

1. ARM Limited. Cortex-A8 technical reference manual, 2006–2010. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/DDI0344K_cortex_a8_r3p2_trm.pdf
2. ARM Limited. ARM architecture reference manual: ARMv7-A and ARMv7-R edition, 2007–2014. https://silver.arm.com/download/ARM_and_AMBA_Architecture/AR570-DA-70000-r0p0-00rel2/DDI0406C_C_arm_architecture_reference_manual.pdf
3. ARM Limited. ARM Cortex-A9 technical reference manual, 2008–2016. http://infocenter.arm.com/help/topic/com.arm.doc.100511_0401_10_en/arm_cortexa9_trm_100511_0401_10_en.pdf
4. Bernstein, D.J.: Curve25519: new Diffie-Hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 207–228. Springer, Heidelberg (2006). doi:10.1007/11745853_14
5. Bernstein, D.J.: Batch binary Edwards. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 317–336. Springer, Heidelberg (2009). doi:10.1007/978-3-642-03356-8_19

6. Bernstein, D.J., Birkner, P., Joye, M., Lange, T., Peters, C.: Twisted Edwards curves. In: Vaudenay, S. (ed.) AFRICACRYPT 2008. LNCS, vol. 5023, pp. 389–405. Springer, Heidelberg (2008). doi:10.1007/978-3-540-68164-9_26
7. Bernstein, D.J., Chuengsatiansup, C., Lange, T., Schwabe, P.: Kummer strikes back: new DH speed records. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014. LNCS, vol. 8873, pp. 317–337. Springer, Heidelberg (2014). doi:10.1007/978-3-662-45611-8_17
8. Bernstein, D.J., Lange, T.: eBACS: ECRYPT Benchmarking of Cryptographic Systems. http://bench.cr.yp.to/results-dh.html. Accessed 15 May 2016
9. Bernstein, D.J., Schwabe, P.: NEON crypto. In: Prouff, E., Schaumont, P. (eds.) CHES 2012. LNCS, vol. 7428, pp. 320–339. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33027-8_19
10. Câmara, D., Gouvêa, C.P.L., López, J., Dahab, R.: Fast software polynomial multiplication on ARM processors using the NEON engine. In: Cuzzocrea, A., Kittl, C., Simos, D.E., Weippl, E., Xu, L. (eds.) CD-ARES 2013. LNCS, vol. 8128, pp. 137–154. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40588-4_10
11. Chou, T.: Sandy2x: new Curve25519 speed records. In: Dunkelman, O., Keliher, L. (eds.) SAC 2015. LNCS, vol. 9566, pp. 145–160. Springer, Cham (2016). doi:10.1007/978-3-319-31301-6_8
12. Costello, C., Longa, P.: Four$\mathbb{Q}$: four-dimensional decompositions on a $\mathbb{Q}$-curve over the Mersenne prime. In: Advances in Cryptology – ASIACRYPT 2015. LNCS, vol. 9452, pp. 214–235. Springer, Heidelberg (2015). https://eprint.iacr.org/2015/565
13. Costello, C., Longa, P.: Four$\mathbb{Q}$lib (2015). http://research.microsoft.com/en-us/projects/fourqlib/
14. Faz-Hernández, A., Longa, P., Sánchez, A.H.: Efficient and secure algorithms for GLV-based scalar multiplication and their implementation on GLV-GLS curves (extended version). J. Cryptogr. Eng. **5**(1), 31–52 (2015)
15. Faz-Hernández, A., Longa, P., Sánchez, A.H.: Efficient and secure algorithms for GLV-based scalar multiplication and their implementation on GLV-GLS curves. In: Benaloh, J. (ed.) CT-RSA 2014. LNCS, vol. 8366, pp. 1–27. Springer, Cham (2014). doi:10.1007/978-3-319-04852-9_1
16. Gallant, R.P., Lambert, R.J., Vanstone, S.A.: Faster point multiplication on elliptic curves with efficient endomorphisms. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 190–200. Springer, Heidelberg (2001). doi:10.1007/3-540-44647-8_11
17. Gaudry, P., Schost, E.: Genus 2 point counting over prime fields. J. Symb. Comput. **47**(4), 368–400 (2012)
18. Hisil, H., Wong, K.K.-H., Carter, G., Dawson, E.: Twisted Edwards curves revisited. In: Pieprzyk, J. (ed.) ASIACRYPT 2008. LNCS, vol. 5350, pp. 326–343. Springer, Heidelberg (2008). doi:10.1007/978-3-540-89255-7_20
19. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996). doi:10.1007/3-540-68697-5_9
20. Longa, P., Sica, F.: Four-dimensional Gallant-Lambert-Vanstone scalar multiplication. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 718–739. Springer, Heidelberg (2012). doi:10.1007/978-3-642-34961-4_43
21. Naehrig, M., Niederhagen, R., Schwabe, P.: New software speed records for cryptographic pairings. In: Abdalla, M., Barreto, P.S.L.M. (eds.) LATINCRYPT 2010. LNCS, vol. 6212, pp. 109–123. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14712-8_7

22. Robinson, T.: 50 Billion ARM Processors shipped. ARM Connected Community, News Blog (2014). http://armdevices.net/2014/02/26/50-billion-arm-processors-shipped/
23. U.S. Department of Commerce/National Institute of Standards and Technology. Digital Signature Standard (DSS). FIPS-186-4 (2013). http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf