Four$\mathbb{Q}$ on embedded devices with strong countermeasures against side-channel attacks

Zhe Liu[1], Patrick Longa[2], Geovandro Pereira[1],
Oscar Reparaz[3], and Hwajeong Seo[4]

[1] University of Waterloo, Canada
{zhelu.liu,geovandro.pereira}@uwaterloo.ca
[2] Microsoft Research, USA
plonga@microsoft.com
[3] imec-COSIC KU Leuven, Belgium
oscar.reparaz@esat.kuleuven.be
[4] Institute for Infocomm Research (I2R), Singapore
hwajeong84@gmail.com

Abstract. This work deals with the energy-efficient, high-speed and high-security implementation of elliptic curve scalar multiplication, elliptic curve Diffie-Hellman (ECDH) key exchange and elliptic curve digital signatures on embedded devices using Four$\mathbb{Q}$ and incorporating strong countermeasures to thwart a wide variety of side-channel attacks. First, we set new speed records for *constant-time* curve-based scalar multiplication, DH key exchange and digital signatures at the 128-bit security level with implementations targeting 8, 16 and 32-bit microcontrollers. For example, our software computes a static ECDH shared secret in ∼7.0 million cycles (or 0.9 seconds @8MHz) on a low-power 8-bit AVR microcontroller which, compared to the fastest Curve25519 and genus-2 Kummer implementations on the same platform, offers 2x and 1.4x speedups, respectively. Similarly, it computes the same operation in ∼559 thousand cycles on a 32-bit ARM Cortex-M4 microcontroller, achieving a factor-2.5 speedup when compared to the fastest Curve25519 implementation targeting the same platform. A similar speed performance is observed in the case of digital signatures. Second, we engineer a set of side-channel countermeasures taking advantage of Four$\mathbb{Q}$'s rich arithmetic and propose a secure implementation that offers protection against a wide range of sophisticated side-channel attacks, including differential power analysis (DPA). Despite the use of strong countermeasures, the experimental results show that our Four$\mathbb{Q}$ software is still efficient enough to outperform implementations of Curve25519 that only protect against timing attacks. Finally, we perform a differential power analysis evaluation of our software running on an ARM Cortex-M4, and report that no leakage was detected with up to 10 million traces. These results demonstrate the potential of deploying Four$\mathbb{Q}$ on low-power applications such as protocols for the Internet of Things.

Keywords. Elliptic curves, Four$\mathbb{Q}$, ECDH, digital signatures, embedded devices, IoT, efficient implementation, energy efficiency, constant-time, side-channel attacks, strong countermeasures.

1 Introduction

By 2020, it is estimated that about 50 billion devices or "things" will be connected to the Internet [22]. While this explosive growth of the so-called "Internet of Things" (IoT) promises to revolutionize the way in which the world interacts and works, many experts agree that the unprecedented level of connectivity is going to bring not only enormous benefits and innovative applications but also highly challenging problems, especially regarding security and privacy [41, 58].

Elliptic curve cryptography (ECC) is a popular public-key system that has become an attractive candidate to enable strong cryptography on constrained devices. Its reduced key sizes and great performance are nicely matched by its solid security foundation based on the elliptic curve discrete logarithm problem (ECDLP). Hence, it is foremost relevant to research

ECC-based mechanisms that could ameliorate efficiency and power limitations with the goal of making ECC suitable for constrained applications.

Costello and Longa's Four$\mathbb{Q}$ curve [18], a special instance of the endomorphism-based constructions studied by Smith [64] and Guillevic and Ionica [31], is a high-performance elliptic curve that provides about 128 bits of security and enables efficient and secure scalar multiplications. Implementations based on this curve have been shown to achieve the fastest computations of variable-base, fixed-base and double scalar multiplications to date on a large variety of x64 and ARMv7–A processors [18, 45]. In addition, Järvinen et al. recently reported the first Four$\mathbb{Q}$-based hardware implementation and set a speed record for curves over large prime characteristic fields on FPGAs [36]. Overall, results obtained from different software and hardware platforms consistently show that Four$\mathbb{Q}$ is more than 5 times faster than the standardized NIST curve P-256 and more than 2 times faster than Curve25519.

This performance trait is especially attractive for IoT, when devices need to keep clock frequencies to a minimum (in order to fulfill limited power budgets) and yet need to minimize the impact on the device's response time. Moreover, Four$\mathbb{Q}$'s high speed is expected to have a direct positive impact in energy savings, since reduced computing time typically translates to lower energy consumption. Even though IoT devices often possess a few KB of RAM and may provide even less than 100KB of flash memory, for many battery-powered devices *energy* is by far the most precious resource. In some applications such as wireless sensor networks, devices must last for long periods of time on a single battery charge. For these cases, Four$\mathbb{Q}$ appears to be a promising candidate to enable strong public-key cryptography.

Side-channel attacks. Protection against side-channel attacks [40, 39] represents another important aspect of the security in embedded devices. These attacks, which have been the focus of intense research since Kocher' seminal paper [40], can be classified as: *passive* attacks (a.k.a. side-channel analysis (SCA)), such as differential side-channel analysis (DSCA) [39], timing [40], correlation [6], collision [27] and template [8] attacks, among many other variants; and *active* attacks (a.k.a. fault attacks). Refer to [3, 24] for detailed taxonomies of attacks and countermeasures. Certainly, most of these attacks can be rendered ineffective (or greatly limited in impact) by restricting the lifespan of secrets, for instance, by using *fully* ephemeral ECDH key exchange[1]. However, some protocols such as those based on static ECDH or ephemeral ECDH with cached public keys can be subjected to these attacks and, thus, might require additional defenses. In this work, we focus on *passive attacks*.

Our contributions. We present the first implementations of Four$\mathbb{Q}$-based scalar multiplication, ECDH key exchange and Schnorr-type signatures on 8, 16, and 32-bit microcontrollers (MCUs), and demonstrate that this curve can deliver the fastest curve-based computations on embedded IoT devices, potentially helping to achieve stringent design goals in terms of response time and energy (see §3 and §4). For example, a static ECDH shared key is computed 2x, 1.8x, and 2.5x faster than the fastest Curve25519 implementations on 8-bit AVR, 16-bit MSP430X, and 32-bit ARM Cortex-M4 MCUs, respectively. Notably, the speedup ratios increase to 2.8x, 2.5x and 3.4x, respectively, when considering the case of fully ephemeral ECDH. Similarly, signing and verification using Schnorr$\mathbb{Q}$—a Schnorr-type, Four$\mathbb{Q}$-based signature scheme proposed by Costello and Longa [20]—are computed 2.3x and 1.9x faster

[1] In some contexts, the term "ephemeral ECDH" is used even when public keys are cached and reused for a certain period of time. We stress that using fresh private and public keys per each key exchange (which we refer to as "fully ephemeral ECDH") greatly increases resilience against side-channel attacks and limits the attack surface to essentially *one* connection.

(resp.) than the genus-2 Kummer signature scheme by Renes et al. [56] on an 8-bit AVR microcontroller.

Further, we analyze how these efficiency improvements translate to significantly lower energy costs on a popular wireless sensor node called MICAz, which contains an 8-bit AVR ATmega microcontroller. For example, we show that a static ECDH shared key computation using Four$\mathbb{Q}$ with 64-byte public keys demands 28.41mJ (or 236,676 total computations for the life of a double AA battery). This represents a factor-2 reduction in energy when compared with the Curve25519 implementation from [23] (the cost is 56,68mJ per shared key computation using Curve25519, or 119,089 computations when using a double AA battery; see §4). Notably, we show that these energy savings are achieved while still keeping a competitive memory footprint for suitably chosen parameters (see Table 6 in §4), which makes Four$\mathbb{Q}$ especially attractive for embedded applications.

In addition, we present, to the best of our knowledge, the first *publicly-available* design and implementation of an elliptic curve-based system that includes defenses against a wide variety of passive attacks (see §5). Our protected scalar multiplication, ECDH and signature algorithms, which include a set of efficient countermeasures that have been especially tailored for Four$\mathbb{Q}$, are designed to minimize the risk of timing attacks, simple and differential side-channel analysis (SSCA/DSCA), correlation and collision attacks, and specialized attacks such as the doubling attack [27], the refined power attack (RPA) [29], zero-value point attacks (ZVP) [1], same value attacks (SVA) [48], exceptional procedure attacks [35], invalid point attacks [5], and small subgroup attacks. To assess the soundness of our algorithms, we carry out a differential power analysis evaluation on an STM32F4Discovery board containing a popular ARM Cortex-M4 MCU. We perform leakage detection tests and correlation power analysis attacks to verify that indeed the implemented countermeasures substantially increase the required attacker effort for unprofiled vertical attacks (see §6).

Previous works in the literature presenting protected ECC implementations only include basic countermeasures against a subset of the attacks we deal with in this paper [71, 50]. Moreover, reported implementations (other than implementations exclusively protected against timing attacks [23]) have not been publicly released. Our software for ARM Cortex-M4 has been made publicly available as part of the Four$\mathbb{Q}$lib library [19]:

`https://github.com/Microsoft/FourQlib`.

Likewise, the implementations for AVR and MSP are available at:

`https://github.com/geovandro/microFourQ-AVR`, and
`https://github.com/geovandro/microFourQ-MSP`.

Disclaimer. No software implementation is able to guarantee 100% side-channel security. In some cases, certain powerful attacks such as template attacks [8] can be carried out using a single target trace, making any randomization or masking technique useless [52]. Moreover, the issue gets more complicated for embedded devices that lack access to a good source of randomness. Since many SCA attacks closely depend on the underlying hardware, it is recommended to include additional countermeasures at the software and hardware levels depending on the targeted platform. Also, note that hardware countermeasures are usually required to properly deal with most sophisticated invasive attacks.

Organization. The paper is organized as follows. In §2, we cover the basics about Four$\mathbb{Q}$, as well as the ECDH key exchange and digital signature schemes targeted in this work. Then,

we describe implementation details for AVR, MSP430X and ARM Cortex-M4 MCUs in §3. In §4, we present benchmarking results as well as the analysis of the energy cost on the MICAz wireless sensor node. In §5, we describe the proposed countermeasures and our side-channel protected algorithms for scalar multiplication, ECDH key exchange and Schnorr$\mathbb{Q}$ signatures. Finally, the side-channel security analysis of our protected implementation for ARM Cortex-M4 is presented in §6.

2 Preliminaries: Four$\mathbb{Q}$

Four$\mathbb{Q}$, introduced by Costello and Longa in 2015 [18], is defined by the complete twisted Edwards [4] equation $\mathcal{E}/\mathbb{F}_{p^2} : -x^2 + y^2 = 1 + dx^2y^2$, where the quadratic extension field $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ for $i^2 = -1$ and $p = 2^{127} - 1$, and $d = 125317048443780598345676279555970305165 \cdot i + 42058576488057777768770$. The prime order subgroup $\mathcal{E}(\mathbb{F}_{p^2})[N]$, where N is the 246-bit prime corresponding to $\#\mathcal{E}(\mathbb{F}_{p^2}) = 392 \cdot N$, is used to carry out cryptographic computations. In this subgroup, the neutral element is given by $\mathcal{O}_\mathcal{E} = (0, 1)$ and the inverse of a point (x, y) is given by $(-x, y)$.

Four$\mathbb{Q}$ is equipped with *two* efficiently computable endomorphisms, ψ and ϕ, which give rise to four-dimensional decompositions. The computation of a constant-time, exception-free variable-base scalar multiplication with the form $[m]P$, where m is an integer in $[1, 2^{256})$ and P is a point from $\mathcal{E}(\mathbb{F}_{p^2})[N]$, then proceeds as follows (see Algorithm 1). First, one needs to prepare a precomputed table with the eight points $T[u] = P + u_0\phi(P) + u_1\psi(P) + u_2\phi(\psi(P))$ for $0 \leq u \leq 7$, where $u = (u_2, u_1, u_0)_2$, at Steps 1 and 2, to then execute the scalar decomposition and multiscalar recoding algorithms at Steps 3 and 4. As defined before, let a scalar m be any integer in the range $[1, 2^{256})$. Four$\mathbb{Q}$'s decomposition procedure [18, Proposition 5] maps m to a set of *multiscalars* $(a_1, a_2, a_3, a_4) \in \mathbb{Z}^4$ such that $0 \leq a_i < 2^{64}$ for $i = 1, ..., 4$ and such that a_1 is odd. These multiscalars are then recoded using [18, Algorithm 1] to a representation consisting of exactly 65 "signed digit-columns" d_j and "sign masks" m_j for $j = 0, ..., 64$ (pseudocodes for Four$\mathbb{Q}$'s decomposition and multiscalar recoding algorithms can be found in Appendix A). Finally, the evaluation stage (Steps 5–7) consists of an initial point loading and a single loop of 64 iterations, where each iteration computes one doubling and one addition with the point from $T[\,]$ corresponding to the current digit-column.

Efficient implementations of the curve arithmetic in Algorithm 1 are based on variants of the *extended twisted Edwards coordinates* $(X : Y : Z : T)$ from [33], where $T = XY/Z$ and $Z \neq 0$. In particular, our constant-time implementations for AVR, MSP430X and Cortex-M4 use the coordinate strategy described in [18, §5.2], which consists of four point representations: $\mathbf{R_1} : (X, Y, Z, T_a, T_b)$, 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 $\mathbf{R_4} : (X, Y, Z)$. In the scalar multiplication's main loop, point doublings are computed as $\mathbf{R_1} \leftarrow \mathbf{R_4}$ and point additions as $\mathbf{R_1} \leftarrow \mathbf{R_1} \times \mathbf{R_2}$ (precomputed points are stored using $\mathbf{R_2}$). Note that converting point addition results from $\mathbf{R_1}$ to $\mathbf{R_4}$ (as required by inputs to point doublings) is for free: one simply ignores coordinates T_a and T_b.

2.1 Encoding and parsing integers and points

Next, we describe the encoding and parsing of integers and elliptic curve points closely following [20]. These definitions will be used later in the ECDH and Schnorr$\mathbb{Q}$ routines.

Algorithm 1 Four$\mathbb{Q}$'s scalar multiplication on $\mathcal{E}(\mathbb{F}_{p^2})[N]$ (from [18]).

Input: Point $P \in \mathcal{E}(\mathbb{F}_{p^2})[N]$ and integer scalar $m \in [0, 2^{256})$.
Output: $[m]P$.

Compute endomorphisms and precompute lookup table:
1: Compute $\phi(P)$, $\psi(P)$ and $\psi(\phi(P))$.
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 [18, Prop. 5]. See Listing 1.1, App A.
4: Recode (a_1, a_2, a_3, a_4) into $(d_{64}, \ldots, d_0)$ and $(m_{64}, \ldots, m_0)$ using [18, Alg. 1]. See Listing 1.2, App A. 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: $\quad Q = [2]Q + s_i \cdot T[d_i]$
8: **return** Q

Encoding and parsing integers. An integer $S \in \{0, 1, \ldots, N-1\}$ is encoded in little-endian form as the 256-bit string $\underline{S} = (S_0, S_1, \ldots, S_{255})$. This bit string is parsed to the integer $S = S_0 + 2S_1 + \cdots + 2^{255}S_{255}$. Note that the most significant bits $S_{246}, S_{247}, \ldots, S_{255}$ are 0 since N is a 246-bit prime.

Encoding elements in $\mathbb{F}_{p^2}$. An element $y = a + b \cdot i \in \mathbb{F}_{p^2}$ is encoded in little-endian form as $\underline{y} = (a_0, \ldots, a_{126}, 0, b_0, \ldots, b_{126})$, which is defined as "negative" if and only if $a_{126} = 1$, or if $b_{126} = 1$ and $a = 0$.

We now define the following functions for compression and decompression of points.

- Compress(P): given an input point $P = (x, y) \in \mathcal{E}$, this function encodes P as the 256-bit string $\underline{P} = \underline{(x, y)}$, which is the 255-bit encoding of y followed by a sign bit; this sign bit is 1 if and only if x is negative.
- Expand($\underline{S}$): given a 256-bit input string $\underline{S}$, this function recovers $P = (x, y)$ as follows: parse the first 255 bits as y, compute $u/v = (y^2 - 1)/(dy^2 + 1)$, and compute $\pm x = \sqrt{u/v}$, where the $\pm$ is chosen so that the sign of x matches the 256-th bit of the string $\underline{S}$.

Refer to [20, Appendix A] and [42, Appendix B] for low-level details about the point decompression procedure for the Expand() function.

2.2 Cofactor elliptic curve Diffie-Hellman key exchange

In this section, we describe the ECDH key exchange using Four$\mathbb{Q}$ in *two* variants: (i) using 64-byte public keys, and (ii) using compressed 32-byte public keys. Let's first define the following function denoted by "DH" [42]:

function DH(m, P)
$\quad$ **if** $P \notin \mathcal{E}$ **then return** failed
$\quad Q = [392]P$
$\quad T = [m]Q$
$\quad$ **if** $T = (0, 1)$ **then return** failed
$\quad$ **return** T in affine coordinates

end function

Note that the function DH validates the input point P against the curve equation in order to thwart invalid point attacks. The multiplication by 392, which is not required to be computed in constant-time, clears the cofactor and guarantees that the point Q belongs to $\mathcal{E}(\mathbb{F}_{p^2})[N]$, as required by Alg. 1 for the computation of $[m]Q$. This measure protects against small subgroup attacks.

For the remainder, assume that the generator $G = (G_x, G_y)$ is given by [20]:

$$\begin{aligned} G_x &= 348322423331659341519764392731774944 42 + \\ &\quad 400395300848778818162862150379150028 70 \cdot i \\ G_y &= 189411461867937157347740481657941326 15 + \\ &\quad 146361984425930646555497992424795179868 \cdot i \end{aligned}$$

An ECDH key exchange with 64-byte public keys can then be carried out as follows. Two users, Alice and Bob, pick random integers m_A and m_B (resp.) in the range $[0, 2^{256})$, and then compute the public keys $A = [m_A]G$ and $B = [m_B]G$ (resp.), where G is the generator. After exchanging public keys, Alice computes $K_A = \text{DH}(m_A, B)$ and Bob computes $K_B = \text{DH}(m_B, A)$. The y-coordinate of the value $K = K_A = K_B$ can then be used as the shared secret.

Let a public key be represented by the coordinates (x, y) for which $x = a + bi$ and $y = c + di \in \mathbb{F}_{p^2}$. Before computing the shared secret, each user should verify that the values a, b, c, d of the received public key are $< 2^{127}$.

ECDH key exchange with 32-byte public keys. It is possible to reduce the size of the public keys to only 32 bytes using point compression and the Compress/Expand functions defined in §2.1. In this case, the ECDH key exchange mechanism proceeds as follows [42]. Alice and Bob pick random integers m_A and m_B (resp.) in the range $[0, 2^{256})$, and then compute the public keys $\underline{A} = \text{Compress}([m_A]G)$ and $\underline{B} = \text{Compress}([m_B]G)$ (resp.). After exchanging public keys, Alice computes $K_A = \text{DH}(m_A, \text{Expand}(\underline{B}))$ and Bob computes $K_B = \text{DH}(m_B, \text{Expand}(\underline{A}))$. As before, the y-coordinate of the value $K = K_A = K_B$ is the shared secret.

Before decompressing a sender's public key $\underline{S}$ using the Expand() function, the user should verify that $\underline{S}_{127}$ (i.e., the 128-th bit of the received public key) is 0.

Cost. The cost of *static* ECDH is dominated by the scalar multiplication required by the computation $\text{DH}([m]P)$ for a given public key P. This scalar multiplication is computed using Algorithm 1 in our software. In the case of ephemeral ECDH, the cost of calculating the public key (via a scalar multiplication of the form $[m]G$) should also be taken into account. Since the generator G is known in advance, calculating a public key can be sped up using a *fixed-base* scalar multiplication. In our implementations we use the modified LSB-set comb method [25, Alg. 5] for this computation.

2.3 SchnorrQ signature scheme

SchnorrQ [20] is a *deterministic* digital signature scheme that is based on the well-known Schnorr signature scheme [60]. It was designed closely following the EdDSA signature specifications [44] and using the elliptic curve FourQ.

Let H be a hash function with 512-bit output and G be the generator defined above. Let k be a 256-bit secret key. The pseudocode of the SchnorrQ public key generation function is as follows.

function *SchnorrQ_generate*(k)
  $h = H(k)$
  $s = (h_0, h_1, \ldots, h_{255}) \pmod N$
  $A = [s]G$
  return $\underline{A} = \text{Compress}(A)$
end function

The function above outputs the public key $\underline{A}$ (which is the encoding of the point A) corresponding to the secret key k.

Let M be the message to be signed. The pseudocodes of the signing and verification functions are detailed below.

function *SchnorrQ_sign*$(k, \underline{A}, M)$
  $h = H(k)$
  $s = (h_{256}, h_{257}, \ldots, h_{511})$
  $r = H(s, M) \pmod N$
  $R = [r]G$
  $\underline{R} = \text{Compress}(R)$
  $S = r - s \cdot H(\underline{R}, \underline{A}, M) \pmod N$
  return $(\underline{R}, \underline{S})$
end function

The output of the function above is the signature $(\underline{R}, \underline{S})$, a 64-byte string in which $\underline{R}$ is the encoding of point R and $\underline{S}$ is the encoding of the integer $S \in \{0, 1, \ldots, N-1\}$.

function *SchnorrQ_verify*$(\underline{R}, \underline{S}), \underline{A}, M)$
  $A = \text{Expand}(\underline{A})$
  if ($\underline{A}_{127} \neq 0$ or $\underline{R}_{127} \neq 0$ or $A \notin \mathcal{E}$ or $S \geq 2^{246}$) **then return** reject
  $h = H(\underline{R}, \underline{A}, M) \pmod N$
  $\underline{R'} = \text{Compress}([S]G + [h]A)$
  if ($\underline{R} = \underline{R'}$) **then return** accept
  return reject
end function

If the verification function above is successful the signature is accepted. Otherwise, it is rejected. The function includes a few checks to make sure that the inputs are valid and are in the permitted range: the non-imaginary values of the encoded y-coordinate of $\underline{A}$ and $\underline{R}$ must be in the range $[0, 2^{127})$ and S must be an integer in the range $[0, 2^{246})$. As can be seen, these values are not expected to be fully reduced, i.e., they are not required to be in the proper ranges $[0, 2^{127} - 1)$ and $[0, N)$, respectively. This is done in order to simplify the checks. Checking that the point $A = (x, y)$ lies on $\mathcal{E}$ can be carried out by verifying that the equation $-x^2 + y^2 = 1 + dx^2y^2$ holds.

Cost. The most costly operation during signing is the computation $R = [r]G$. Since G is known in advance, calculating R can be sped up using a fixed-base scalar multiplication. As

for the case of ECDH, our implementations use the modified LSB-set comb method [25, Alg. 5] for this operation. In the case of verification, the cost is dominated by the computation $[S]G+[h]A$. Typically, this operation is efficiently computed via a *simultaneous* double scalar multiplication algorithm using interleaving with width-w NAFs. Since G is known in advance, the number of operations that are needed for computing the $[S]G$ part can be reduced by increasing the corresponding window size w. In our implementations we induce four-dimensional decompositions on both S and h to then carry out a fast *eight-way* multiscalar multiplication.

3 Implementation details on AVR, MSP and ARM

In this section, we summarize the most relevant implementation aspects for *three* popular microcontrollers: 8-bit AVR ATmega, 16-bit MSP430X, and 32-bit ARM Cortex-M4.

First, we give some details about the underlying arithmetic over $\mathbb{F}_{(2^{127}-1)^2}$.

3.1 Implementation of arithmetic over $\mathbb{F}_{(2^{127}-1)^2}$

In contrast to traditional ECC curves, which are defined over a prime field $\mathbb{F}_p$, Four$\mathbb{Q}$ is defined over the quadratic extension field $\mathbb{F}_{p^2}$ for $p = 2^{127} - 1$. Let $a = a_0 + a_1 \cdot i, b = b_0 + b_1 \cdot i \in \mathbb{F}_{p^2}$. Operations in $\mathbb{F}_{p^2}$ are computed as follows

$$\begin{aligned}
a + b &= (a_0 + b_0) + (a_1 + b_1) \cdot i, \\
a - b &= (a_0 - b_0) + (a_1 - b_1) \cdot i, \\
a \times b &= (a_0 \cdot b_0 - a_1 \cdot b_1) + ((a_0 + a_1) \cdot (b_0 + b_1) - a_0 \cdot b_0 - a_1 \cdot b_1) \cdot i, \\
a^2 &= (a_0 + a_1) \cdot (a_0 - a_1) + (2a_0 \cdot a_1) \cdot i, \\
a^{-1} &= a_0 \cdot (a_0^2 + a_1^2)^{-1} - a_1 \cdot (a_0^2 + a_1^2)^{-1} \cdot i,
\end{aligned}$$

where operations on the right are carried out in $\mathbb{F}_p$. Naïvely, multiplication requires three integer multiplications, three modular reductions, two field additions and three field subtractions, whereas squaring requires only two integer multiplications, two modular reductions, two field additions and one field subtraction.

We improve the performance of multiplication and squaring in $\mathbb{F}_{p^2}$ by transforming field additions into simple integer additions. This is possible because our integer multiplication accepts inputs in the extended range $[0, 2^{128})$. For the case of Cortex-M4, we speed up multiplication in $\mathbb{F}_{p^2}$ by exploiting lazy reduction, which allows the elimination of one modular reduction by delaying the reductions of the products until the very end of the computation.

Field inversions $a^{-1} \pmod p$ are computed via Fermat's Little Theorem as $a^{p-2} \pmod p$, using a fixed multiplication-and-squaring chain with 126 field squarings and 10 field multiplications in order to have a constant-time execution.

Modular reduction is particularly efficient on Four$\mathbb{Q}$. Let $r = a + b$ be the result of adding two operands in $\mathbb{F}_p$. To reduce this result, one only needs to reset the 128-th bit of r and then perform an addition between that top bit and the updated value of r, i.e., given $0 \leq r < 2 \cdot (2^{127} - 1)$, compute $a + b \pmod p$ as $r \pmod{2^{127}} + (r \gg 127)$. For example, assume that the intermediate result r of the addition is stored in the 16 AVR registers `r0:r15`. Then, modular reduction can be efficiently implemented using AVR assembly as follows

$$\texttt{MOV r16, r15} \rightarrow \texttt{ANDI r15, 0x7F} \rightarrow \texttt{ADD r16, r16} \rightarrow \texttt{ADC r0, 0} \rightarrow \cdots \rightarrow \texttt{ADC r15, 0}$$

A similar procedure applies to reductions after multiplications and squarings, with the difference that reduction is, in these cases, applied to an intermediate result with double precision (i.e., 32 bytes). Specifically, given an input $0 \leq r < (2^{127}-1)^2$, the fast reduction algorithm requires two consecutive rounds computing $r \leftarrow r \pmod{2^{127}} + (r \gg 127)$.

We remark that our implementations of the field and extension field arithmetic run in constant-time, i.e., do not use secret-dependent branches.

3.2 Implementation on 8-bit AVR ATmega

Many widely-used low-cost smartcards and wireless sensor nodes are equipped with 8-bit AVR microcontrollers; e.g., the MICAz mote. AVR microcontrollers, such as the Atmel ATmega128 or ATxmega256A3, have an 8-bit RISC instruction set and a modified Harvard architecture that features 32 8-bit general-purpose registers denoted by `r0:r31`. From this pool of registers, the last three pairs, called X (`r27:r26`), Y (`r29:r28`), and `Z` (`r31:r30`), are used as 16-bit address pointers to load and store data from memory. The AVR instruction set supports a total of 133 instructions, and each instruction has a fixed latency; for example, ordinary arithmetic/logical instructions such as addition (`ADD`) and addition with carry (`ADC`) are executed in a single clock cycle, while unsigned multiplication (`MUL`) as well as load/store instructions take two clock cycles.

For our benchmarks, we used the IAR Embedded Workbench – AVR 6.80.7, which features an assembler and a cycle-accurate graphical simulator, and targeted the ATxmega256A3 model. This specific microcontroller has 256KB of programmable flash memory, 16KB of SRAM and 4KB of EEPROM, and operates at a maximum frequency of 32MHz.

Since our algorithms are designed to be constant-time, every execution of scalar multiplication is expected to take the same number of cycles independently of the input values. We confirmed this by monitoring many executions of a single operation of our implementation.

Finite field operations. For the 128-bit integer multiplication, we use 2-level Karatsuba in a recursive way, as described in [34]. At the higher level, 128-bit multiplication uses one-level Karatsuba based on 3 64-bit multiplications. At the lower level, each 64-bit multiplication consists of 3 32-bit multiplications in one-level Karatsuba, and each of these 32-bit multiplications employs product-scanning multiplication. Thanks to the short size of operands and intermediate results, almost all of them fit in general purpose registers (only one byte needs to be stored in the stack). For the 128-bit squaring, we employ the Sliding Block Doubling (SBD) method [61] rather than Karatsuba [34], since SBD has been shown to achieve better performance in the case of short 128-bit operands. To achieve an efficient modular reduction, we integrated both integer multiplication/squaring and modular reduction at the assembly level. This reduces the number of load and store instructions by the length of intermediate results. Modular reduction over the prime $p = 2^{127}-1$, as well as the arithmetic over $\mathbb{F}_{p^2}$, was implemented as described in §3.1.

The benchmarking results of our implementation of the $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ functions are compiled in Table 1.

3.3 Implementation on 16-bit MSP430X

The ultra-low power MSP430X is a representative 16-bit microcontroller [21] that includes support for 27 core instructions and 16 registers (`r0:r15`). It also includes an external 16-bit or 32-bit hardware multiplier that operates in parallel to the CPU. The multiplier offers

Table 1. Cycle counts for field and quadratic extension field operations on 8-bit AVR (including function-call overheads).

fp_add	fp_sub	fp_mul	fp_sqr	fp_inv
155	159	1,598	1,026	150,535

fp2_add	fp2_sub	fp2_mul	fp2_sqr	fp2_inv
384	385	5,758	3,622	156,171

three different modes: MPY (unsigned multiplication), MPYS (signed multiplication) and MAC (unsigned multiply-and-accumulate). In general, other instructions take one cycle when working with general-purpose registers. This cost may increase depending on the instruction format and addressing mode.

In our benchmarks, we targeted the MSP430FR5969 model, which is suitable for use in wireless sensor nodes. This MCU features 2KB of SRAM and 64KB of FRAM (code) memory, and operates at up to 16MHz. We followed the same methodology for cycle count acquisition that was employed for AVR using the IAR Embedded Workbench (MSP430 6.50.1).

Finite field operations. We make extensive use of the 16-bit MAC operation available in the targeted MSP430X microcontroller. This operation, which computes $16 \times 16 + 32 \rightarrow 33$-bit, was used as basic block to realize a 128-bit integer multiplication in a column-wise way [30]. Squaring was implemented using the SBD method, as in the case of AVR. Modular reduction, as well as the arithmetic over $\mathbb{F}_{p^2}$, was implemented as described in §3.1. In our implementation, we reduced the number of memory accesses by performing operand caching using the multiplier mapped memory, which enables multiple MAC computations without the need of accessing memory [43].

The benchmarking results of our implementation of the $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ functions are compiled in Table 2. It is worth pointing out that we include function calls and returns in our cycle counts, which are usually disregarded in the literature.

Table 2. Cycle counts for field and quadratic extension field operations on 16-bit MSP430X using 16-bit multiplier (including function-call overheads).

fp_add	fp_sub	fp_mul	fp_sqr	fp_inv
102	101	1,027	927	131,819

fp2_add	fp2_sub	fp2_mul	fp2_sqr	fp2_inv
233	231	3,624	2,391	135,315

3.4 Implementation on 32-bit ARM Cortex–M4

Cortex-M4 [2] is part of the increasingly popular ARM Cortex-M family, which includes a wide range of 32-bit RISC ARM microcontrollers that are successfully penetrating the embedded and mobile markets. It supports the ARMv7E-M instruction set, which comprises Thumb-2 instructions and additional saturating/SIMD instructions called the "DSP extension". The Cortex-M4 architecture has a 3-stage pipeline with branch speculation, includes 16 32-bit registers (`r0:r15`), and supports a mix of 16 and 32-bit operations corresponding

to Thumb-2. Instructions that are relevant for our implementation include 32-bit arithmetic and logical instructions such as addition (`ADD`), addition with carry (`ADC`), as well as memory instructions that perform a single-data loading/storing (`LDR/STR`) or multiple-data loading/storing (`LDM/STM`). Field arithmetic can take advantage of the powerful single-cycle multiply and multiply-and-accumulate instructions from the DSP extension: `UMUL`, `UMLAL`, and `UMAAL`. These instructions compute the product 32×32-bit $\rightarrow$ 64-bit (`UMUL`), plus a 64-bit accumulation with a single 64-bit value (`UMLAL`) or plus a 64-bit accumulation with two 32-bit values (`UMAAL`).

To evaluate the performance of our implementation, we use an STM32F4Discovery board [67] that contains a 32-bit ARM Cortex-M4F STM32F407VGT6 microcontroller [66]. This MCU has 1MB of flash memory, 192KB of SRAM and 64KB of CCM (core coupled memory) data RAM, and can be clocked at a frequency of up to 168MHz. Compilation was performed with the GNU ARM Embedded toolchain and GNU GCC v4.9.2.

Finite Field Operations. We leveraged the power of MAC instructions to realize an efficient and compact multiplication using the schoolbook method. Let A and B be two field elements represented with 32-bit limbs $A[0], ..., A[3]$ and $B[0], ..., B[3]$, respectively, and assume that B is pre-loaded in registers `r3:r6`. The computations $A[0] \times B, \ldots, A[3] \times B$ are performed word-wise as follows

```
LDR   r7,[r1]
UMULL r9,r8,r7,r3 → UMLAL r8,r10,r7,r4 → ··· → UMLAL r11,r12,r7,r6
LDR   r7,[r1,#4]  → STR   r9,[r13],#4
UMLAL r8,r9,r7,r3 → UMAAL r9,r10,r7,r4 → ··· → UMAAL r11,r12,r7,r6
LDR   r7,[r1,#8]  → STR   r8,[r13],#4
UMLAL r9,r8,r7,r3 → UMAAL r8,r10,r7,r4 → ··· → UMAAL r11,r12,r7,r6
LDR   r7,[r1,#12] → STR   r9,[r13],#4
UMLAL r8,r9,r7,r3 → UMAAL r9,r10,r7,r4 → ··· → UMAAL r11,r12,r7,r6
```

The result of the integer multiplication consists of the lowest 32-bit terms produced after each of the first three MAC sequences (which are stored in the stack) plus the final values in registers `r8:r12`. In order to complete a field multiplication one can execute the modular reduction described in §3.1. However, in the case of multiplication in $\mathbb{F}_{p^2}$ we do much better by applying lazy reduction on a basic schoolbook multiplication that computes $a \times b$ as $(a_0 \cdot b_0 - a_1 \cdot b_1) + (a_0 \cdot b_1 + a_1 \cdot b_0) \cdot i$ for elements $a = a_0 + a_1 \cdot i, b = b_0 + b_1 \cdot i \in \mathbb{F}_{p^2}$. For 32-bit platforms, this is expected to be more efficient than Karatsuba, because it avoids the overhead introduced by the extra operations. Our implementation of squaring over $\mathbb{F}_{p^2}$ also takes advantage of the fast multiplication described above. The computation follows the approach described in §3.1, i.e., we compute a^2 as $(a_0 + a_1) \cdot (a_0 - a_1) + (2a_0 \cdot a_1) \cdot i$. In this case, performance can be improved further by noting that the two field additions do not need modular corrections since our multiplication algorithm works for any input in $[0, 2^{128})$. Thus, in total squaring in $\mathbb{F}_{p^2}$ requires two integer multiplications, two reductions, two integer additions and one field subtraction.

Tables 3 compiles the experimental results for extension field operations, which were obtained on the STM32F4Discovery board using the maximum core frequency of 168MHz. For each case, we averaged the timings of 10^7 iterations running the same function (or 10^5 iterations in the case of inversion).

Table 3. Cycle counts for quadratic extension field operations on 32-bit ARM Cortex-M4 (including function-call overheads).

fp2_add	fp2_sub	fp2_mul	fp2_sqr	fp2_inv
84	86	358	215	21,056

4 Results and analysis of constant-time implementations

In this section, we summarize implementation results for 8-bit AVR, 16-bit MSP430X, and 32-bit ARM Cortex-M4 microcontrollers. For our experiments, we targeted an ATxmega256A3 MCU for the case of AVR, and an MSP430FR5969 MCU for the case of MSP. In both cases, implementations were compiled and evaluated using the IAR Embedded Workbench. For the remainder of this section, these microcontrollers are assumed to be clocked at 8MHz. For the case of ARM Cortex-M4, we targeted the STM32F407VGT6 MCU using an STM32F4Discovery board. The microcontroller was clocked at 168MHz, and compilation was performed with the GNU ARM Embedded toolchain and GNU GCC v4.9.2.

Our Four$\mathbb{Q}$ implementations are based on Algorithm 1 for the case of variable-base scalar multiplication. For the case of fixed-base scalar multiplication, we use the modified LSB-set comb method from [25, Alg. 5], which requires a table with $v \cdot 2^{w-1}$ points (the parameters v and w denote the number of internal tables and their window size, respectively). For the case of double scalar multiplication (required during Schnorr$\mathbb{Q}$ verifications), we first decompose the two scalars in two sets of multiscalars, recode them using width-w NAF and then apply an eight-way multiscalar multiplication. This computation requires the selection of the window size parameters w_P and w_Q corresponding to the precomputation tables required by the fixed and variable bases, respectively. In particular, the parameter w_P fixes the size of the *offline* table to $4 \times 2^{w_P-2}$ points. The computation of double scalar multiplication is not required to be carried out in constant time in the context of signature verification.

The implemented algorithms guarantee regular, exception-free execution (see §2) and run in constant-time. Hence, they are protected against timing and exceptional procedure attacks. Note that cache attacks do not apply to the targeted AVR ATxmega MCU, since its architecture does not support the use of cache memory. On the other hand, MSP430FRxxxx MCUs present some form of integrated caching which is activated when the MCU operates at a higher frequency than the access frequency of the FRAM [69]. Specifically, in the MSP430FR family, the FRAM can be operated at up to 8 MHz without use of this cache. Since we fix the frequency at 8MHz, our software runs in constant-time with no risk of timing leakage. We note that our implementation includes an option (selected at build time) to use code that is secure in the presence of cache memory (when higher operating frequencies are used). Finally, in the targeted STM32F4 MCU we disable the data cache memory by clearing the `DCEN` bit in the flash access control register `FLASH_ACR` [68] during executions of variable-base and fixed-base scalar multiplications.

Even though the risk of cache attacks is avoided, we still need to guarantee that computations over points from the precomputed table have a constant flow. In particular, points extracted at Step 7 of Alg. 1 are always negated and the correct values are masked out using logical operations.

At the high-level, we implemented the ECDH schemes described in §2.2, which are protected against invalid point and small subgroup attacks. Likewise, the Schnorr$\mathbb{Q}$ verification routine (see §2.3) includes the necessary checks to avoid invalid inputs.

In summary, our basic implementations reported in this section are protected against timing and cache attacks, simple side-channel analysis, exceptional procedure attacks, invalid point and small subgroup attacks.

Results for scalar multiplication and ECDH. The results for the three targeted microcontrollers are summarized in Table 4. For comparison, we include two efficient alternatives that have been deployed on various microcontrollers: the "μKummer" key exchange implementation by Renes et al. [56] using the genus-2 Kummer surface by Gaudry and Schost [28], and the "Curve25519" implementations by Düll et al. [23] and De Santis et al. [59]. The Kummer surface enables fast static DH key exchange with a small footprint. However, it does not support efficient, exception-free fixed-base algorithms which inject a significant speedup in settings such as ephemeral DH key exchange, signature key generation and signing. μKummer's DH public keys are also 50% larger (compared to options that use 32-byte public keys). In the case of Curve25519, although this curve supports efficient fixed-base computations via its isomorphic Edwards form, Curve25519 implementations typically target static ECDH and, thus, do not offer this optimization option (as is the case of [23] and [59]).

Table 4 includes results for variable-base and fixed-base scalar multiplication, static ECDH and fully ephemeral ECDH key exchange. For ECDH on Four$\mathbb{Q}$, we evaluate the use of both 32 and 64-byte public keys. As we previously argued, the use of fully ephemeral ECDH (i.e., using a fresh public key per key exchange) increases the security of the protocol and provides superior protection against side-channel attacks. For applications in which this is not possible, we comment that a "relaxed" ephemeral ECDH computation would have roughly the same cost of a static ECDH.

As can be seen, our Four$\mathbb{Q}$-based implementations set new speed records for scalar multiplication and ECDH by a large margin on all of the targeted platforms. In particular, for variable-base computations, Four$\mathbb{Q}$ is 2.1x, 1.9x, and 2.7x faster than Curve25519 on AVR, MSP430X, and Cortex-M4, respectively. These results are roughly the same when considering static ECDH. Similarly, for the case of ephemeral ECDH our implementations are between 2.4x and 3.4x faster than Curve25519 implementations without fixed-base support. When compared against μKummer on AVR, Four$\mathbb{Q}$ achieves roughly factor-1.4 speedup for computing variable-base scalar multiplication and static ECDH. This gap has a significant increase to factor-2 speedup when considering the case of ephemeral ECDH. Note that the Kummer surface has not been implemented on MSP430X and Cortex-M4 MCUs.

Table 4 also show that the performance cost of reducing the size of public keys from 64 to 32 bytes is between 3% and 8%. This poses a trade-off between computing/energy costs (which are lower for 64 bytes) and memory consumption/network delay (which are lower for 32 bytes).

Results for signatures. The performance of Schnorr$\mathbb{Q}$ on the three targeted microcontrollers, including the cost of hashing for the case of short messages of 64 bytes, is summarized in Table 5. For comparison, we include the "μKummer" signature scheme by Renes et al. [56], which is based on the scheme by Chung et al. [9], and the "Ed25519" implementation by Nascimento et al. [50].

Table 5 shows that Four$\mathbb{Q}$ supports extremely fast signatures: signature generation and verification are 2.3x and 1.9x faster (resp.) than μKummer on AVR. As can be seen, the performance gap is especially large for signing, since genus-2 Kummer lacks support for efficient,

Table 4. Performance (in cycles) of scalar multiplication and ECDH operations on 8-bit AVR ATmega, 16-bit MSP430X, and 32-bit ARM Cortex-M4 microcontrollers for different state-of-the-art implementations. In our implementations, we set $v = w = 5$, which fixes the size of the offline table used for ephemeral ECDH to 80 points or 7.5KB of memory. Cycle counts are rounded to the nearest 10^2 cycles.

Source	scalar multiplication		ECDH	
	fixed-base	var-base	static	ephemeral
8-bit AVR ATmega				
Curve25519 [23]	13,900,400[1]	13,900,400	13,900,400[3]	27,800,800[2,3]
μKummer [56]	9,513,500[1]	9,513,500	9,739,100[4]	19,945,200[2,4]
Four$\mathbb{Q}$ (this work)	**3,007,300**	**6,600,700**	**6,980,700**[5] **7,315,200**[3]	**9,988,100**[5] **10,322,600**[3]
16-bit MSP430X (16-bit multiplier) @8MHz				
Curve25519 [23]	7,933,300[1]	7,933,300	7,933,300[3]	15,866,600[2,3]
Four$\mathbb{Q}$ (this work)	**1,851,300**	**4,280,400**	**4,527,900**[5] **4,826,100**[3]	**6,379,200**[5] **6,677,400**[3]
32-bit ARM Cortex-M4				
Curve25519 [59]	1,423,700[1]	1,423,700	1,423,700[3]	2,847,400[2,3]
Four$\mathbb{Q}$ (this work)	**279,800**	**530,300**	**559,200**[5] **606,500**[3]	**838,400**[5] **885,800**[3]

[1] Montgomery ladder is used for fixed-base and variable-base scalar multiplication.
[2] Estimated, since authors only provided counts for static ECDH.
[3,4,5] Public key sizes are 32, 48 and 64 bytes, resp.

exception-free fixed-base algorithms, as previously mentioned (on the other hand, μKummer signatures are very compact with 48 bytes in size). Likewise, in comparison with the Ed25519 implementation from [50], our implementation carries out signature generation and verification 4.2x and 3.5x faster (resp.) on AVR. We are not aware of μKummer and Ed22519 implementations for MSP430X and Cortex-M4 MCUs.

Memory usage. Table 6 summarizes the memory consumption of our implementations and compares it with other works. It presents results individually for static ECDH, ephemeral ECDH (with both 32 and 64-byte public keys), signature generation and signature verification. For each case, we report memory and speed figures for different table sizes (i.e., for different values of w, v and w_P).

As can be seen, there is plenty of flexibility to choose from in between the fastest and the most memory-efficient alternatives. Notably, options that are competitive with other works in terms of memory do not exhibit a significant lose in performance. For example, our signature generation code on AVR demands less memory than μKummer when switching from a table with $w = v = 5$ to $w = v = 4$, and still keeps a significant speedup ratio: only suffers a relatively modest decrease from 2.3x to 2.0x.

Our implementations, with exception of Cortex-M4, also occupy less memory than state-of-the-art Curve25519 implementations [23] in the case of static ECDH. While ephemeral ECDH requires more memory, its use is optional and can be opted out if the memory space is constrained. In this regard, our Four$\mathbb{Q}$-based software offers great flexibility by allowing to choose between parameters that are memory-friendly and parameters that minimize computing time and energy costs.

Table 5. Performance (in cycles) of signature operations on 8-bit AVR ATmega, 16-bit MSP430X, and 32-bit ARM Cortex-M4 microcontrollers for different state-of-the-art implementations. In our implementations, we set $v = w = 5$, which fixes the size of the offline table used for key and signature generation to 80 points or 7.5KB of memory, and set $w_P = 8$ and $w_Q = 4$, which fixes the size of the offline table used for signature verification to 256 points or 24KB of memory. The cost of hashing is included (messages are assumed to be 64 bytes long). Cycle counts are rounded to the nearest 10^2 cycles.

Source	key gen.	signature gen.	signature ver.
8-bit AVR ATmega			
Ed25519 [50][1,4]	-	19,047,700	30,776,900
μKummer [56][2,5]	10,206,200	10,404,000	16,240,500
Schnorr$\mathbb{Q}$ (this work)[3,4]	**3,416,500**	**4,515,700**	**8,671,200**
16-bit MSP430X (16-bit multiplier) @8MHz			
Schnorr$\mathbb{Q}$ (this work)[3,4]	**1,906,500**	**2,136,100**	**5,443,000**
32-bit ARM Cortex-M4			
Schnorr$\mathbb{Q}$ (this work)[1,4]	**317,900**	**478,900**	**732,900**

[1,2,3] SHA-512, SHAKE-128 and Blake2b are used for hashing, resp.
[4,5] Signature sizes are 64 and 48 bytes, resp.

4.1 Analysis of energy cost: case study with AVR

In general, the total energy cost of a protocol consists of two parts, namely, the cost of computing a cryptographic operation and the wireless network communication. The energy cost of a cryptographic protocol can be estimated by using the energy per cycle and the total number of cycles of the computation; on the other hand, the communication cost comprises the cost of transmission and reception. For our analysis below, we consider the case of a MICAz wireless sensor node containing an 8-bit AVR ATmega128L MCU.

According to [54], the energy cost per cycle of a computation on the MICAz sensor node is roughly 4.07nJ. Recalling that our Four$\mathbb{Q}$ implementation of the ephemeral ECDH using 32-byte public keys has an execution time of approximately $10.332 \cdot 10^6$ clock cycles on AVR, then the energy cost amounts to $\mathrm{W}_e = 4.07\mathrm{nJ} \cdot 10.332 \cdot 10^6 \simeq 42.05\mathrm{mJ}$. From the estimate in [54] the per-bit energy cost for transmission and reception (via ZigBee transceiver) requires 0.209μJ and 0.226μJ on the MICAz running at 7.37MHz. In this case, the overhead for transmitting the 32-byte public key of one node would be $\mathrm{W}_t = 256 \cdot 0.209\mu\mathrm{J} \simeq 0.053\mathrm{mJ}$, and the energy cost for reception is roughly $\mathrm{W}_r = 256 \cdot 0.226\mu\mathrm{J} \simeq 0.057\mathrm{mJ}$. Therefore, the total energy consumption of performing an ECDH key exchange is $\mathrm{W} = \mathrm{W}_e + \mathrm{W}_t + \mathrm{W}_r \simeq 42.16\mathrm{mJ}$. As stated in [54], the MICAz family sensor nodes are usually powered by double AA battery cells, and require a supply voltage higher than 2.7V. Thus, the node could only use 31.25% of the total capacity. In this case, the energy capacity available for supplying a MICAz family is only $6,750\mathrm{W}$. Therefore, we can perform $6,750/0.04216 \simeq 160,104$ ECDH key exchanges before the batteries die.

After doing a similar analysis for Curve25519, the energy consumption for this curve is estimated at $\mathrm{W} = \mathrm{W}_e(113.14\mathrm{mJ}) + \mathrm{W}_t(0.053\mathrm{mJ}) + \mathrm{W}_r(0.058\mathrm{mJ}) \simeq 113.25\mathrm{mJ}$, which means that one can compute up to $6,750/0.11325 \simeq 59,602$ ECDH key exchanges with the same battery budget. For μKummer, the total energy consumption is $\mathrm{W} = \mathrm{W}_e(77.43\mathrm{mJ}) + \mathrm{W}_t(0.080\mathrm{mJ}) + \mathrm{W}_r(0.086\mathrm{mJ}) \simeq 77.59\mathrm{mJ}$, or $6,750/0.07759 \simeq 86,995$ ECDH key exchanges. This means that our Four$\mathbb{Q}$ implementation on AVR is able to run roughly 2.7x and 1.8x more ephemeral

Table 6. Memory consumption (in bytes) and performance (in cycles) of ECDH and signature operations on 8-bit AVR ATmega, 16-bit MSP430X, and 32-bit ARM Cortex-M4 microcontrollers for different state-of-the-art implementations. In the case of Cortex-M4, our implementation was compiled with the option `-finline-limit=10` to reduce the memory consumption. Cycle counts are rounded to the nearest 10^2 cycles.

Source	Function	Parameters	Memory code + data	Memory stack	Performance
8-bit AVR ATmega					
Curve25519 [23]	static ECDH	-	17,710	494	13,900,400[2]
	ephem. ECDH				27,800,800[1,2]
μKummer [56][6]	static ECDH	-	$> 9{,}490$	429	9,739,100[3]
	ephem. ECDH	-		812	19,945,200[1,3]
	signature gen.	-	$> 16{,}516$	926	10,404,000
	signature ver.	-		992	16,240,500
FourQ/SchnorrQ[5] (this work)	static ECDH	-	15,088 + 888	2,714	6,980,700[4]
		-	16,538 + 928	2,715	7,315,200[2]
	ephem. ECDH	$w = 4, v = 4$	22,412 + 984	2,714	10,785,500 [4]
			23,552 + 1,024	2,715	11,120,000 [2]
		$w = 5, v = 5$	27,052 + 984	2,714	9,988,100 [4]
			28,192 + 1,024	2,715	10,322,600 [2]
	signature gen.	$w = 4, v = 4$	11,602 + 204	1,590	5,175,400
		$w = 5, v = 5$	16,242 + 204	1,593	4,515,700
	signature ver.	$w_P = 2$	24,638 + 928	5,050	11,467,900
		$w_P = 6$	30,458 + 928	5,050	9,155,100
		$w_P = 8$	48,802 + 928	5,050	8,671,200
16-bit MSP430X (16-bit multiplier) @8MHz					
Curve25519 [23]	static ECDH	-	13,112	384	7,933,300[2]
	ephem. ECDH				15,866,600[1,2]
FourQ/SchnorrQ[5] (this work)	static ECDH	-	10,714 + 784	2,754	4,527,900 [4]
		-	11,764 + 792	2,754	4,826,100 [2]
	ephem. ECDH	$w = 4, v = 4$	17,516 + 784	2,754	6,904,000 [4]
			18,344 + 792	2,754	7,202,100 [2]
		$w = 5, v = 5$	22,188 + 784	2,754	6,379,200 [4]
			23,016 + 792	2,754	6,677,400 [2]
	signature gen.	$w = 4, v = 4$	9,780 + 4	1,590	2,660,900
		$w = 5, v = 5$	14,452 + 4	1,608	2,136,100
	signature ver.	$w_P = 2$	18,294 + 792	5,026	7,060,500
		$w_P = 6$	24,070 + 792	5,026	5,612,800
		$w_P = 8$	42,558 + 792	5,026	5,443,000
32-bit ARM Cortex-M4: ***medium code optimization***					
Curve25519 [59]	static ECDH	-	3,750	740	1,423,700[2]
	ephem. ECDH				2,847,400[1,2]
FourQ/SchnorrQ[5] (this work)	static ECDH	-	6,736 + 640	-	579,700[4]
		-	7,644 + 640	-	629,800[2]
	ephem. ECDH	$w = 4, v = 4$	12,740 + 640	-	929,200[4]
			13,368 + 640	-	977,600[2]
		$w = 5, v = 5$	17,340 + 640	-	863,400[4]
			17,968 + 640	-	913,600[2]
	signature gen.	$w = 4, v = 4$	8,616	-	541,400
		$w = 5, v = 5$	13,220	-	475,600
	signature ver.	$w_P = 2$	9,448 + 640	-	950,200
		$w_P = 6$	15,240 + 640	-	771,600
		$w_P = 8$	33,664 + 640	-	747,900

[1] Estimated, since authors only provided counts for static ECDH.
[2,3,4] Public key sizes are 32, 48 and 64 bytes, resp.
[5,6] Signature sizes are 64 and 48 bytes, resp.

ECDH key exchanges with the same battery budget. We remark that this relative energy efficiency increases if we consider our ECDH option with 64-byte public keys.

Let's now consider the case of static ECDH using our uncompressed option with 64-byte public keys. In this case, Four$\mathbb{Q}$ demands 28.41mJ (or 236,676 total executions when using a double AA battery). This is 2x lower energy than the case of Curve25519 (56,68mJ per shared key computation, or 119,089 computations for the life of a double AA battery) and 1.4x lower energy than the case of μKummer (39,81mJ per shared key computation, or 169,555 computations).

5 Side-channel countermeasures

This section begins with a description of countermeasures especially tailored for Four$\mathbb{Q}$. Then, we present our protected scalar multiplication algorithm, cover implementation aspects of table lookups and describe protected ECDH key exchange and digital signature schemes. Finally, we discuss the rationale behind our protected algorithms.

5.1 Specialized side-channel countermeasures for Four$\mathbb{Q}$

The use of randomization, if done properly, greatly increases the effort needed to perform DSCA and other similar attacks, both in terms of data complexity (number of measurements needed [7]) and computational effort (time to perform the attack [57]). While a randomized implementation does not completely eliminate all the leakage information, it potentially makes side-channel attacks significantly harder and more expensive by forcing attackers to take an impractical number of measurements and use sophisticated techniques.

In an ECC scalar multiplication operation there is ample room for randomization of internal computations, especially on curves such as Four$\mathbb{Q}$ because of its rich underlying mathematical structure. Coron proposed *three* randomization techniques to protect ECC against DPA attacks: scalar randomization, point blinding and projective coordinate randomization [17]. Other popular methods include key splitting [11], and random curve and field isomorphisms [37].

Next, we describe especially-tailored scalar randomization and point blinding techniques optimized for use with Four$\mathbb{Q}$.

Scalar randomization. The typical approach is to randomize the scalar m by adding a multiple of the curve order $\#E$ using a random value r, i.e., computing $m' = m + r \cdot \#E$. It is well known that this randomization can be ineffective if the prime p has a special structure [51, 10, 11, 63]. Indeed, when p is a pseudo-Mersenne prime with the form $2^k - c$ for small c, by Hasse's theorem the binary representation of the top half of the curve order $\#E$ consists of either only 1's or a 1 followed by 0's and, thus, the most significant bits of $m + r\#E$ are those of m. As consequence, the random value r must be greater than $\approx k/2$ as a minimum requirement, which means that the cost of protected implementations of curves such as Curve25519 increase by at least 50% when using this countermeasure.

We avoid this significant performance degradation by specializing the GLV-based scalar randomization by Ciet et al. [12] to Four$\mathbb{Q}$. Our explicit countermeasure is described below.

Proposition 1 (Scalar Randomization). *Let the multiscalars* $(a'_1, a'_2, a'_3, a'_4) = (a_1, a_2, a_3, a_4) + \mathbf{c}$ *be the decomposition result of a given integer* m*, as defined in [18, Prop. 5], where*

$\mathbf{c} = 5\mathbf{b}_2 - 3\mathbf{b}_3 + 2\mathbf{b}_4$ *is a vector in the lattice of zero decompositions* $\mathcal{L}$ *and* $\mathbf{B} = (\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3, \mathbf{b}_4)$ *is the Babai optimal basis in [18, Prop. 3]. Let* $\mathbf{V} = (\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4) = (\mathbf{b}_2 - \mathbf{b}_3 + \mathbf{b}_4, 2\mathbf{b}_2 - \mathbf{b}_3 + \mathbf{b}_4, \mathbf{b}_1 + \mathbf{b}_2 + \mathbf{b}_4, \mathbf{b}_1 + 2\mathbf{b}_2 - \mathbf{b}_3 + \mathbf{b}_4)$ *be a matrix of four independent vectors in* $\mathcal{L}$ *such that* $||\mathbf{v}_i||_\infty < 2^{62}$ *for* $i = 1, \ldots, 4$*, and let* $\mathbf{r} = (r_1, r_2, r_3, r_4)$ *be a vector with random integer elements in* $[0, 2^{16})$*. Then, the multiscalar set* $(a'_1, a'_2, a'_3, a'_4) + \mathbf{r} \cdot (\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4)$ *is a valid decomposition of* m *with all four randomly-generated coordinates less than* 2^{80}*.*

Proof. From [18, Prop. 5], we have that the multiscalar $(a_1, a_2, a_3, a_4) = (m, 0, 0, 0) - \sum_{i=1}^{4} \tilde{\alpha}_i \mathbf{b}_i$, where $\tilde{\alpha}_i = \lfloor \alpha_i \rceil - \epsilon_i$ with $\epsilon_i \in \{0, 1\}$ for $i = 1, 2, 3, 4$ and $(\alpha_1, \alpha_2, \alpha_3, \alpha_4) \in \mathbb{Q}^4$ is the unique solution to $(m, 0, 0, 0) = \sum_{i=1}^{4} \alpha_i \mathbf{b}_i$ in [18, Prop. 4]. Since $\mathbf{c} \in \mathcal{L}$ then $(a'_1, a'_2, a'_3, a'_4) = (a_1, a_2, a_3, a_4) + \mathbf{c}$ is in $(m, 0, 0, 0) + \mathcal{L}$, which shows that it is a valid decomposition of m. Also, $(a'_1, a'_2, a'_3, a'_4) = \sum_{i=1}^{4} (\alpha_i - (\lfloor \alpha_i \rceil - \epsilon_i))\mathbf{b}_i + \mathbf{c}$ is in $\mathcal{P}_\epsilon(\mathbf{B}) + \mathbf{c}$, for the parallelepiped $\mathcal{P}_\epsilon(\mathbf{B}) := \{\mathbf{Bx} \,|\, \mathbf{x} \in [-1/2, 3/2)^4\}$ defined in [18, §4.2]. All four coordinates of $\mathcal{P}_\epsilon(\mathbf{B}) + \mathbf{c}$ are positive and less than 2^{64}. In a similar fashion, since $\mathbf{r} \cdot \mathbf{V} \in \mathcal{L}$, then $(a'_1, a'_2, a'_3, a'_4) + \mathbf{r} \cdot \mathbf{V}$ is also a valid decomposition of m. Let the 4-cube $\mathcal{H}_{\text{ext}} = \{2^{80} \cdot \mathbf{x} \,|\, \mathbf{x} \in [0, 1]^4\}$. All sixteen corners of $\mathcal{P}_\epsilon(\mathbf{B}) + \mathbf{c} + \mathbf{r} \cdot \mathbf{V}$ are inside the convex body of $\mathcal{H}_{\text{ext}}$, which means that they have all four coordinates positive and less than 2^{80}. □

Proposition 1 specifies the countermeasure procedure with $4 \times 16 = 64$ bits of randomization. This brings enough entropy to provide security against several attacks, especially when combined with additional countermeasures (see §5.2), while requiring a relatively low overhead in comparison with other curves (the cost of Four$\mathbb{Q}$'s scalar multiplication is only increased by 25% in this case). Other applications might try different trade-offs between security and performance by selecting a different length for the random values r_i.

Point blinding. The typical approach is to compute $[m]P$ as $[m](P+R) - S$ for a randomly-generated secret point R and a precomputed point $S = [m]R$. To avoid the cost of an extra scalar multiplication, Coron suggests that R and S are updated at each new execution using $R = [(-1)^b 2]R$ and $S = [(-1)^b 2]S$ for a random bit b. Nevertheless, the method still requires storage for two points and the computation of a full scalar multiplication if the value of m is changed.

It is possible to do better using the extended-binary-based-method with RIP (called "EBRIP") due to Mamiya et al. [46]. In this case, $[m]P$ is computed as $([m]P + R) - R$ using a random point R. The value in parenthesis is computed by splitting m in t portions of equal length and running a t-way simultaneous scalar multiplication in which R is represented as $[(1\bar{1}\bar{1}\ldots\bar{1})_2]R$.

Adapting EBRIP to Four$\mathbb{Q}$ is straightforward: it suffices to assume $t = 4$ and adjust the precomputed values which, in the case of Four$\mathbb{Q}$, use the endomorphisms. The details are shown in Algorithm 2. The overhead of the method is small: the number of precomputations increases from 8 to 16 points (adding 8 extra point additions to the cost), and a final correction subtracting R is required at the end of scalar multiplication.

We note that typical update functions for blinding points offer poor randomization, making them an easy target of collision-like attacks [51, 27]. We improve resilience against these attacks with an inexpensive change to the new update function $R = [(-1)^b 3]R$ for a random bit b.

Algorithm 2 SCA-protected Four$\mathbb{Q}$'s scalar multiplication on $\mathcal{E}(\mathbb{F}_{p^2})[N]$.

Input: Point $P = (x_P, y_P)$, blinding point $R = (x_R, y_R) \in \mathcal{E}(\mathbb{F}_{p^2})[N]$, integer scalar m and random value $s \in [0, 2^{256})$, a random bit b, and random values $[r_{81}, r_{80}, \ldots, r_0] \in \mathbb{F}_p^{82}$.
Output: $[m]P$ and updated point R.

Randomize input points and update blinding point R:
1: Set $R = (r_{81} \cdot x_R, r_{81} \cdot y_R, r_{81})$.
2: Compute $R = [(-1)^b 3]R$.
3: Set $P = (r_{80} \cdot x_P, r_{80} \cdot y_P, r_{80})$.
Compute endomorphisms and precompute lookup table:
4: Compute $\phi(P)$, $\psi(P)$ and $\psi(\phi(P))$.
5: Compute $T[u] = -R + [u_0]P + [u_1]\phi(P) + [u_2]\psi(P) + [u_3]\psi(\phi(P))$ for $u = (u_3, u_2, u_1, u_0)_2$ in $0 \leq u \leq 15$. Write $T[u]$ in coordinates (X, Y, Z).
Scalar decomposition, randomization and recoding:
6: Decompose m into the multiscalar (a_1, a_2, a_3, a_4) as in [18, Prop. 5]. See Listing 1.1, App A.
7: Randomize (a_1, a_2, a_3, a_4) as in Proposition 1 and recode to digit-columns $(d_{79}, \ldots, d_0)$ s.t. $d_i = a_1[i] + 2a_2[i] + 4a_3[i] + 8a_4[i]$ for $i = 0, ..., 79$.
Main loop:
8: $Q = R$
9: **for** $i = 79$ **to** 0 **do**
10: $\quad S = (r_i \cdot X_{T[d_i]}, r_i \cdot Y_{T[d_i]}, r_i \cdot Z_{T[d_i]})$.
11: $\quad Q = [2]Q + S$
12: **return** $(Q - R)$ and R in affine coordinates.

5.2 Protected scalar multiplication

Algorithm 2 details our scalar multiplication routine with SCA countermeasures, including the scalar randomization and point blinding techniques described above. Note that we also make extensive use of projective coordinate randomization [17]. This technique is a form of multiplicative masking: in our case, a non-zero element $r \in \mathbb{F}_{2^{127}-1}$ is applied to points (X, Y, Z) in homogeneous projective coordinates to obtain the equivalent *randomized* tuple $(r \cdot X, r \cdot Y, r \cdot Z)$.

Protected ECDH key exchange. In order to use Algorithm 2, the function DH described in §2.2 only needs minor changes and the inclusion of a blinding point B. We assume that a fresh blinding point is generated during key generation, and the value passed and updated each time the protected ECDH function is invoked. The modified function is shown below.

function DH_SCA(m, P, B)
$\quad$ **if** $P, B \notin \mathcal{E}$ **then return** failed
$\quad Q = [392]P$
$\quad T = [m]Q$ and update B using Algorithm 2
$\quad$ **if** $T = (0, 1)$ **then return** failed
$\quad$ **return** T and B in affine coordinates
end function

The function DH_SCA can be directly used in place of the function DH in the ECDH key exchange schemes using 32 and 64-byte public keys that were described in §2.2. As explained before, these functions are protected against invalid point and small subgroup attacks.

Protected Schnorr$\mathbb{Q}$ signature generation. Signature schemes typically have several sensitive variables suitable to attack with DPA. In the case of Schnorr$\mathbb{Q}$, we identify the

following potentially vulnerable computations in the function *SchnorrQ_sign* described in §2.3:

- The scalar multiplication $[r]G$,
- The modular multiplication $s \cdot H(\underline{R}, \underline{A}, M)$, and
- The hash computations $H(k)$ and $H(s, M)$.

Algorithm 2 supports the protected execution of $[r]G$. The blinding point B that is required can be generated during a first call to SchnorrQ's public key generation, and then the value reused and updated every time the signature generation function is invoked. To protect the modular multiplication $s \cdot H(\underline{R}, \underline{A}, M)$, one can apply a simple masking strategy by randomly splitting s into two values s_1 and s_2.

The modified signature generation function is shown below.

function *SchnorrQ_sign_SCA*$(k, \underline{A}, M, B)$
  $h = H(k)$
  $s = (h_{256}, h_{257}, \ldots, h_{511})$
  $r = H(s, M) \pmod N$
  $R = [r]G$ and update B using Algorithm 2
  $\underline{R} = \text{Compress}(R)$
  Pick a random value $s_1 \in [0, \text{mod} N)$
  $s_2 = s - s_1 \pmod N$
  $t_1 = s_1 \cdot H(\underline{R}, \underline{A}, M) \pmod N$
  $t_2 = s_2 \cdot H(\underline{R}, \underline{A}, M) \pmod N$
  $t_1 = r - t_1 \pmod N$
  $S = t_1 - t_2 \pmod N$
  return $(\underline{R}, \underline{S}), B$
end function

Finally, a *masked* implementation of the hash function H is required in order to protect the hash computations $H(k)$ and $H(s, M)$ in the function above (e.g., see [47, 62]).

Reducing table lookup leakage. Table lookups are common to many ECC algorithms (including the proposed routine) and, hence, their secure implementation is crucial. Most works in the literature use *constant-time* table lookups, which simply perform a linear pass over the whole table, masking out the correct result using logical instructions. This masking typically employs masks that are all 0's or 1's, which may be relatively easy to distinguish through SPA. One way to reduce the potential leakage is by using masks with the same Hamming weight. For example, one could use the masking strategy shown below (to extract $T[d]$ from a 16-point table T, as required at Step 10 of Alg. 2).

```
v = 0xAA...A, S ← T[0]   // Table index (d) is between 0 and 15
for i = 1 to 15
   d--   // While d >= 0 mask = 0x55...5, else mask = 0xAA...A
   mask = ((top_bit(d) - 1) & ∼ v)|(∼ (top_bit(d) - 1) & v)
   S ← ((mask & (S^T[i]))^S)^(v & (S^T[i]))
return S = T[d]
```

In this case, the bulk of the extraction procedure is carried out with the new mask values `0x55...5` (used to update `S` with the current table entry) and `0xAA...A` (used to keep the current

value of `S`). Operations over these masks are expected to produce traces that are more difficult to distinguish from each other. Note, however, that this does not eliminate all the potential leakage. For example, a sophisticated attacker might try to reveal the secret digit by observing the operation `(top_bit(d) - 1)` inside the derivation of `mask`, which produces intermediate all-0 or all-1 values. Nevertheless, this operation happens only once per iteration (in contrast to the multiple, word-wise use of the other masks), so the strategy above does reduce the attack surface significantly.

Horizontal DPA attacks could also target operations over the secret digits d_i during table lookups (e.g., at Step 10 of Alg. 2). An attack of this class can observe leakage when manipulating a given secret digit d_j and compare it to d_k under some distinguisher $D(L(d_j), L(d_k))$ to extract information on the relation between d_j and d_k. Our implementation lends itself to mitigations for this: one could apply principles from the table recomputation method to Algorithm 2 to protect the secrets d_i. The implementor can mask the secret digits d_i as $d'_i = d_i \oplus t$ for a random integer $t \in [0, 15)$, and then reorder the table such that $T'[d] = T[d \oplus t]$ for $d = 0, \ldots, 15$. This is repeated for each iteration using as entries the updated table and digits from the preceding iteration. Since the table indexes are changed each time an attacker cannot directly infer what point is being extracted even if the updated digit value is revealed. This reduces further the attack surface if the table reordering is correctly implemented [70].

Finally, another potential attack is to apply a horizontal attack on the table outputs. By default, our routine applies projective coordinate randomization after each point extraction (at Step 10). When horizontal collision-correlation attacks apply, one could reduce the potential leakage by doing a full table randomization at each iteration and before point extraction. This technique should also increase the effectiveness of the countermeasures described above.

Analysis of the protected algorithms. First, it is easy to see that the SCA-protected scalar multiplication in Algorithm 2 inherits the properties of regularity and completeness from Algorithm 1 when using complete twisted Edwards formulas [33]. This means that computations work for any possible input point in $\mathcal{E}(\mathbb{F}_{p^2})$ and any 32-byte scalar string, which thwarts against exceptional procedure attacks [35]. Likewise, [18, Prop. 5] and Proposition 1 lend themselves to constant-time implementations of the scalar decomposition and randomization. This, together with field, extension field and table lookup operations implemented with no secret-dependent branches and no secret-memory addresses, guarantees protection against timing [40] and cache attacks [53]. E.g., refer to §3 for details about our constant-time implementations of the $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$ arithmetic for several MCUs. Additionally, note that the use of regular, constant-time algorithms also protects against SSCA attacks such as SPA [39]. In some platforms, however, some computations might have distinctive operand-dependent power/EM signatures even when the execution flow is constant. Our frequent coordinate randomization and the techniques for minimizing table lookup leakage discussed before should make SSCA attacks exploiting such leakage impractical.

The use of point blinding effectively protects against RPA [29], ZVP [1] and SVA [48] attacks, since the attacker is not able to freely use the input point P to generate special values or collisions during intermediate computations. Poorly-randomized update functions for the blinding point has been the target of collision attacks [27]. We first note that intermediate values in the EBRIP algorithm [46] have the form $R + Q$ or $[2]R + Q$ for some point Q and blinding point R. Therefore, a naïve update function such as $R = [(-1)^b 2]R$ for random bit b allows an attacker to find collisions since an updated blinding value $[2]R$ generates values

that match those of the preceding scalar multiplication. The easy change to the function $R = [(-1)^b 3]R$ at Step 2 of Alg. 2 eliminates the possibility of such collisions, since values calculated with $[3]R$ and $[6]R$ do not appear in a preceding computation.

Our combined use of different randomization techniques, namely randomization of projective coordinates at different stages (Steps 1, 3 and 10), randomization of the scalar and blinding of the base point, injects a high level of randomization to intermediate computations. This is expected to reduce leakage that could be useful to carry out correlation, collision-correlation and template attacks. Moreover, in some cases our especially-tailored countermeasures for Four$\mathbb{Q}$ offer better protection in comparison with other elliptic curves. For example, Feix et al. [26] presents vertical and horizontal collision-correlation attacks on ECC implementations with scalar randomization and point blinding countermeasures. They essentially exploit that randomizing with multiples of the order is ineffective on curves such as the NIST curves and Curve25519, as we explain in §5.1. Our 64-bit scalar randomization does not have this disadvantage and is more cost effective.

As previously discussed, some attacks could target collisions between the precomputed values in Step 5 of Alg. 2 and their secret use at Step 11 after point extraction (for example, using techniques from [32]). One way to increase resilience against this class of attacks is by randomizing the full table before each point extraction using coordinate randomization, and minimizing the attack surface through some clever masking via a linear pass over the full table (this in order to thwart attacks targeting memory accesses [50]). However, other more sophisticated countermeasures might be required to protect against recent one-trace template attacks that inspect memory accesses [49]. We remark that some variants of these attacks are only adequately mitigated at lower abstraction levels, i.e., the underlying hardware architecture should be noisy enough such that these attacks become impractical.

Performance. To assess the performance impact of our countermeasures, we refactored our implementation for ARM Cortex-M4 (§3.4) using the algorithms proposed in this section. In summary, our software computes a static ECDH shared secret in about 1.19 and 1.14 million cycles using 32 and 64-byte public keys, respectively. Therefore, the strong countermeasures induce a roughly 2x slowdown in comparison with the constant-time-only implementation. Notably, these results are still up to 1.25x faster than the fastest constant-time-only Curve25519 results (see Table 4). We comment that, if greater protection is required, adding full table randomization before point extraction at Step 10 of Alg. 2 increases the cost of static ECDH to 2.61 and 2.56 million cycles, respectively.

In the case of Schnorr$\mathbb{Q}$, the performance is only affected during signature generation since verification runs over public data. With our protected algorithm, signature generation's cost increases to 1.36 million cycles, which is still faster than the cost of computing a Curve25519 Montgomery ladder operation that is only protected against timing attacks and does not include other costs such as the hashing. The use of full table randomization before each point extraction in the computation of Alg. 2 increases the cost to 2.78 million cycles. We note that the evaluation of our Schnorr$\mathbb{Q}$ software was carried out with an implementation of SHA-512 that is not protected against side-channel attacks such as DPA.

6 Side-channel evaluation: case study with Cortex–M4

The main goal of the evaluation is to assess the DPA security of the implementation. Our randomization techniques are meant to protect mainly against vertical DPA attacks (cf. [13]

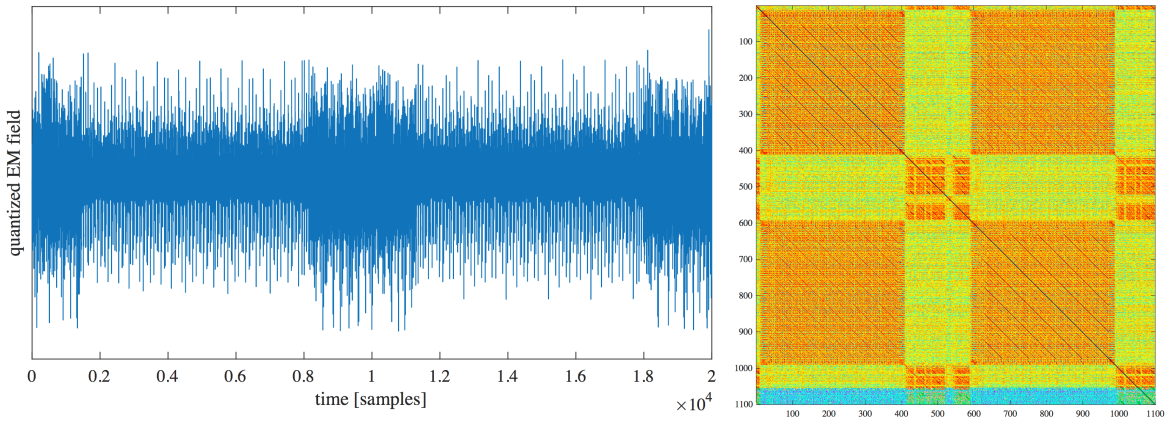

Fig. 1. Left: exemplary EM trace. Right: cross correlation of a single trace.

for this notation). In a vertical DPA attack, the adversary collects many traces corresponding to the multiplication of a known varying input point with a secret scalar. This situation matches, for example, ECDH key exchange protocols. Vertical DPA attacks are probably the easiest to carry out.

Assumptions. We assume that the adversary cannot distinguish values from a single side-channel measurement. In particular, the (small) table indices cannot be retrieved from a single measurement. This assumption is common in practice (cf. [38, §4.1] or [55, §3.1]) and is usually provided by the underlying hardware. Note that masking does not make sense if this assumption is violated, since then it would be trivial to unmask all the required shares to reconstruct the secrets. Masking needs a minimum level of noise to be meaningful [7, 65].

Platform. Our platform is a 32-bit ARM Cortex-M4 STM32F100RB processor with no dedicated security features. We acquire EM traces from a decoupling capacitor in the power line with a Langer RF-5U EM probe and 20 dB amplification. This platform is very low noise: DPA on an unprotected byte-oriented AES implementation succeeds with 15 traces. We give a comfortable setting to the evaluator: he has access to the source code to compute arbitrary predictions and the code contains extra routines for triggering that allow precise alignment of traces.

The EM traces comprise two inner iterations of the main loop (Step 9 in Algorithm 2) as we show in Figure 1.

Methodology. We use two complementary techniques: leakage detection and key-recovery attacks. Failing a leakage detection tests [15, 16, 14] is a necessary, yet not sufficient, condition for key-extracting attacks to work. When an implementation passes a leakage detection test, no data dependency is detected, and hence key-recovery attacks will not work. For key-recovery attacks, we resort to standard CPA attacks [6]; the device behavior is modeled as Hamming weight of register values. As an intermediate targeted sensitive variable we choose the point Q after execution of Step 11 in Algorithm 2. We first test each randomizing countermeasure described in §5.1 in isolation (all others switched off); later the full Algorithm 2 is evaluated. To test the effectiveness of each countermeasure, we first perform the analysis when the countermeasure is switched off. In this situation, a key-recovery attack is expected to work and

a leakage detection test is expected to fail. This serves to confirm that the setup is indeed sound. Then, we repeat the same evaluation when the countermeasure is switched on. The analysis is expected not to show leakage and the CPA attacks are expected to fail. This means that the countermeasure (and *only* it) is effective.

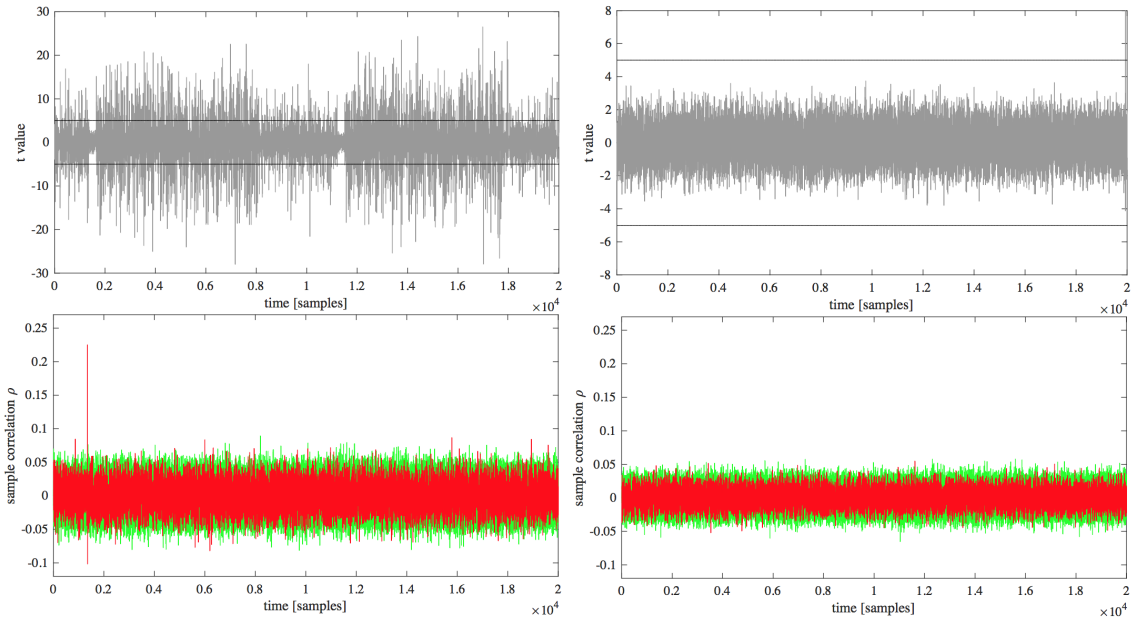

Fig. 2. Top row: fixed-vs-random leakage detection test on the input point. Bottom: CPA attacks. Left column: no countermeasure enabled. Right column: point blinding on/coord. randomization off/scalar randomization off.

No countermeasure. In the first scenario we switch off all countermeasures by fixing the PRNG output to a constant value known to the evaluator. In Figure 2 top left, we plot the result of a non-specific leakage detection test (fix-vs-random on input point) for $5,000$ traces. We can see that the t-statistic clearly exceeds the threshold $C = \pm 4.5$, indicating severe leakage. In Figure 2, bottom left, we plot the result of a key-recovery CPA attack (red for correct subkey hypothesis, green for others). The attack works (sample correlation ρ for the correct subkey hypothesis stands out at $\rho \approx 0.22$).

Point blinding. Here we test the point blinding countermeasure in isolation. We take $5,000$ traces when the point blinding countermeasure is switched on. The evaluator does not know the initial PRNG seed that feeds the masks. In Figure 2, top right, we plot the t-statistic value of the non-specific fix-vs-random leakage detection test on the input point. The t-statistic does not surpass the threshold C. Thus, no first-order leakage is detected.

The results of the CPA attack are in Figure 2, bottom right. The attack does not recover the key, as expected. (In this CPA attack and subsequent ones, the evaluator computes predictions averaging over 2^{10} independent random PRNG seeds, for each subkey hypothesis. This is possible since the evaluator has access to the source code.)

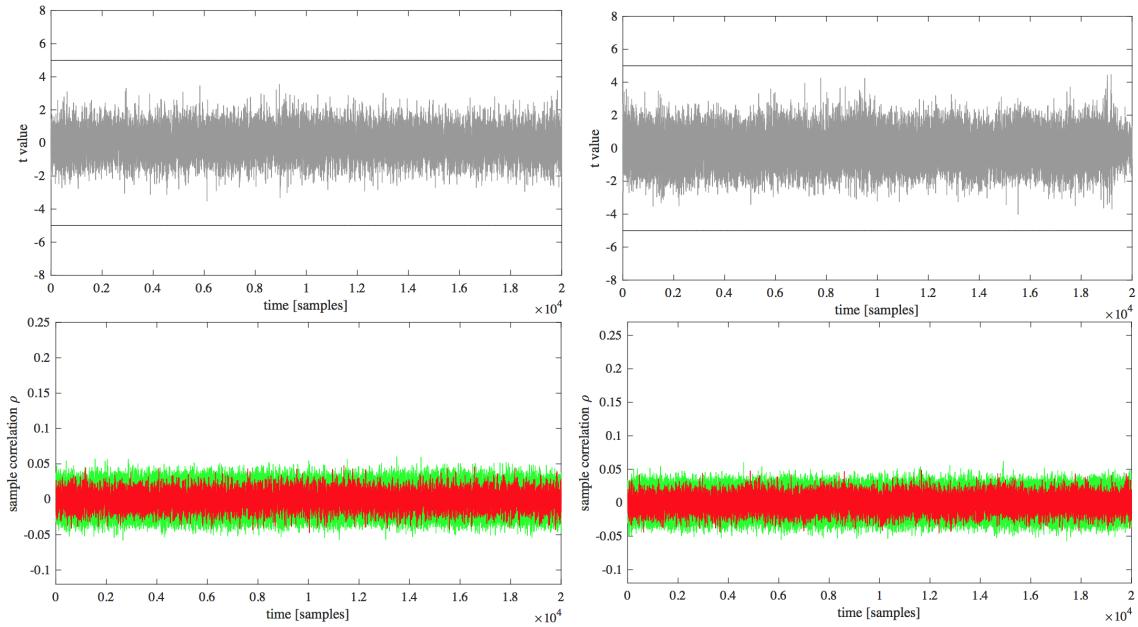

Fig. 3. Left: point blinding off/coord. randomization on/scalar randomization off. Right: point blinding off/-coord. randomization off/scalar randomization on.

Projective coordinate randomization. We use the same test fixture (fix-vs-random on input point) to test the projective coordinate randomization. In Figure 3, top left, we plot the result of the leakage detection test. No first-order leakage is detected. The DPA attack is unsatisfactory as Figure 3, bottom left, shows.

Scalar randomization. Here we perform a fix-vs-random test on the key when the input point is kept fix. In this way, we hope to detect leakages coming from an incomplete randomization of the key. In Figure 3, top right, we plot the result of this leakage detection test. No first-order leakage is detected. For the CPA attack, we keep the key fixed (secret) and vary the input basepoint. The CPA attack does not work, as Figure 3, bottom right, shows.

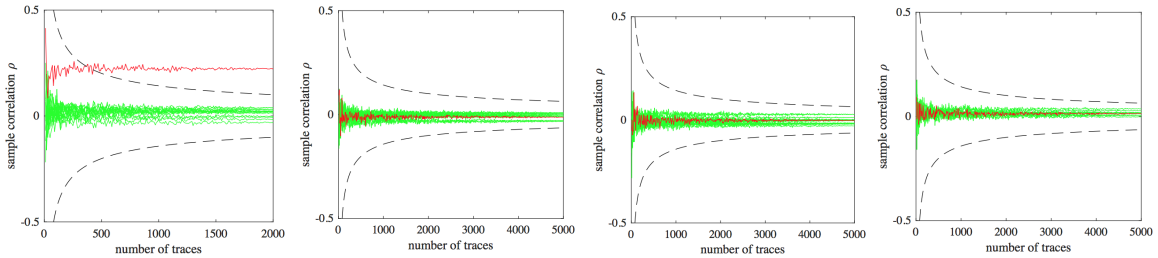

Fig. 4. Evolution of ρ as function of number of traces. Left to right (point blinding/coord. randomization/scalar randomization): off/off/off, on/off/off, off/on/off, off/off/on.

All countermeasures switched on. The implementation is meant to be executed with all the countermeasures switched on. We took 10 million traces and performed a fix-vs-random leakage detection test. No first order leakage was detected.

7 Conclusions

We present the first implementations of the high-performance elliptic curve Four$\mathbb{Q}$ on microcontrollers, and report the fastest timings for curve-based scalar multiplication, Diffie-Hellman key exchange and digital signatures. In particular, our implementation is able to carry out ECDH computations in less than one second for the first time at the 128-bit security level on an AVR microcontroller clocked at 8MHz. Likewise, we report the first results for Schnorr$\mathbb{Q}$ signatures and report signing computations that are more than 2 times faster than the previously fastest computations using μKummer, and about 4 times faster than the fastest Ed25519 implementation in the literature. These efficiency improvements translate not only to reduced latencies but also to significant savings in energy, as we show in our analysis for the MICAz mote. Finally, we propose secure algorithms that offer protection against a wide array of side-channel attacks. To assess the soundness of our algorithms, we carry out a DPA evaluation on an STM32F4Discovery board containing an ARM Cortex-M4 microcontroller. We perform leakage detection tests and correlation power analysis attacks to verify that indeed the implemented countermeasures substantially increase the required attacker effort for unprofiled vertical attacks. The results of this work highlight the potential of using Four$\mathbb{Q}$ for low-power applications such as IoT.

8 Acknowledgments

We would like to thank Craig Costello for helping in the design of the scalar randomization countermeasure, and Pedro R. N. Pedruzzi, Joost Renes and the reviewers for their valuable comments.

References

1. T. Akishita and T. Takagi. Zero-value point attacks on elliptic curve cryptosystem. In C. Boyd and W. Mao, editors, *Information Security, ISC 2003*, volume 2851 of *Lecture Notes in Computer Science*, pages 218–233. Springer, 2003.
2. ARM Limited. Cortex-M4 technical reference manual, 2009–2010. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0439b/DDI0439B_cortex_m4_r0p0_trm.pdf`.
3. R. M. Avanzi. Side channel attacks on implementations of curve-based cryptographic primitives. *IACR Cryptology ePrint Archive, Report 2005/017*, 2005.
4. D. J. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters. Twisted Edwards curves. In S. Vaudenay, editor, *Progress in Cryptology - AFRICACRYPT 2008*, volume 5023 of *Lecture Notes in Computer Science*, pages 389–405. Springer, 2008.
5. I. Biehl, B. Meyer, and V. Müller. Differential fault attacks on elliptic curve cryptosystems. In M. Bellare, editor, *Advances in Cryptology - CRYPTO 2000*, volume 1880 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2000.
6. E. Brier, C. Clavier, and F. Olivier. Correlation power analysis with a leakage model. In M. Joye and J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2004.
7. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. In M. J. Wiener, editor, *Advances in Cryptology - CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

8. S. Chari, J. R. Rao, and P. Rohatgi. Template attacks. In B. S. K. Jr., Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2002.
9. P. N. Chung, C. Costello, and B. Smith. Fast, uniform, and compact scalar multiplication for elliptic curves and genus 2 jacobians with applications to signature schemes. *IACR Cryptology ePrint Archive, Report 2015/983*, 2015.
10. M. Ciet. *Aspects of fast and secure arithmetics for elliptic curve cryptography.* PhD thesis, Université Catholique de Louvain, Louvain-la-Neuve, 2003.
11. M. Ciet and M. Joye. (Virtually) free randomization techniques for elliptic curve cryptography. In S. Qing, D. Gollmann, and J. Zhou, editors, *Information and Communications Security, ICICS 2003*, volume 2836 of *Lecture Notes in Computer Science*, pages 348–359. Springer, 2003.
12. M. Ciet, J. Quisquater, and F. Sica. Preventing differential analysis in GLV elliptic curve scalar multiplication. In B. S. K. Jr., Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 540–550. Springer, 2002.
13. C. Clavier, B. Feix, G. Gagnerot, M. Roussellet, and V. Verneuil. Horizontal correlation analysis on exponentiation. In M. Soriano, S. Qing, and J. López, editors, *Information and Communications Security - ICICS 2010*, volume 6476 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2010.
14. J. Cooper, E. DeMulder, G. Goodwill, J. Jaffe, G. Kenworthy, and P. Rohatgi. Test Vector Leakage Assessment (TVLA) methodology in practice. International Cryptographic Module Conference, 2013.
15. J. Coron, P. C. Kocher, and D. Naccache. Statistics and secret leakage. In Y. Frankel, editor, *Financial Cryptography, FC 2000*, volume 1962 of *Lecture Notes in Computer Science*, pages 157–173. Springer, 2000.
16. J. Coron, D. Naccache, and P. C. Kocher. Statistics and secret leakage. *ACM Trans. Embedded Comput. Syst.*, 3(3):492–508, 2004.
17. J.-S. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 1999*, volume 1717 of *Lecture Notes in Computer Science*, pages 292–302. Springer, 1999.
18. C. Costello and P. Longa. Four$\mathbb{Q}$: Four-dimensional decompositions on a $\mathbb{Q}$-curve over the Mersenne prime. In T. Iwata and J. H. Cheon, editors, *Advances in Cryptology - ASIACRYPT 2015*, volume 9452 of *Lecture Notes in Computer Science*, pages 214–235. Springer, 2015. Full version: `https://eprint.iacr.org/2015/565`.
19. C. Costello and P. Longa. Four$\mathbb{Q}$lib. `https://github.com/Microsoft/FourQlib`, 2015-2017.
20. C. Costello and P. Longa. Schnorr$\mathbb{Q}$: Schnorr signatures on Four$\mathbb{Q}$. *MSR Tech Report*, 2016. Available at: `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/SchnorrQ.pdf`.
21. D. Dang, M. Plant, and M. Poole. Wireless connectivity for the Internet of Things (IoT) with MSP430 microcontrollers (MCUs). Texas Instruments white paper, available at: `http://www.ti.com/lit/wp/slay028/slay028.pdf`, 2014.
22. Dave Evans. The Internet of Things: how the next evolution of the Internet is changing everything , 2011. `http://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf`.
23. M. Düll, B. Haase, G. Hinterwälder, M. Hutter, C. Paar, A. H. Sánchez, and P. Schwabe. High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Designs, Codes and Cryptography*, 77(2-3):493–514, 2015.
24. J. Fan and I. Verbauwhede. An updated survey on secure ECC implementations: Attacks, countermeasures and cost. In D. Naccache, editor, *Cryptography and Security: From Theory to Applications - Essays Dedicated to Jean-Jacques Quisquater on the Occasion of His 65th Birthday*, volume 6805 of *Lecture Notes in Computer Science*, pages 265–282. Springer, 2012.
25. 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 (extended version). *J. Cryptographic Engineering*, 5(1):31–52, 2015.
26. B. Feix, M. Roussellet, and A. Venelli. Side-channel analysis on blinded regular scalar multiplications. In W. Meier and D. Mukhopadhyay, editors, *Progress in Cryptology - INDOCRYPT 2014*, volume 8885 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2014.
27. P. Fouque and F. Valette. The doubling attack - why upwards is better than downwards. In C. D. Walter, Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 269–280. Springer, 2003.
28. P. Gaudry and E. Schost. Genus 2 point counting over prime fields. *J. Symbolic Computation*, 47(4):368–400, 2012.

29. L. Goubin. A refined power-analysis attack on elliptic curve cryptosystems. In Y. Desmedt, editor, *Public Key Cryptography - PKC 2003*, volume 2567 of *Lecture Notes in Computer Science*, pages 199–210. Springer, 2003.
30. C. P. L. Gouvêa and J. López. Software implementation of pairing-based cryptography on sensor networks using the MSP430 microcontroller. In *International Conference on Cryptology in India*, pages 248–262. Springer, 2009.
31. A. Guillevic and S. Ionica. Four-dimensional GLV via the Weil restriction. In K. Sako and P. Sarkar, editors, *Advances in Cryptology - ASIACRYPT 2013*, volume 8269 of *Lecture Notes in Computer Science*, pages 79–96. Springer, 2013.
32. N. Hanley, H. Kim, and M. Tunstall. Exploiting collisions in addition chain-based exponentiation algorithms using a single trace. In K. Nyberg, editor, *Topics in Cryptology - CT-RSA 2015, The Cryptographer's Track at the RSA Conference 2015*, volume 9048 of *Lecture Notes in Computer Science*, pages 431–448. Springer, 2015.
33. H. Hisil, K. K. Wong, G. Carter, and E. Dawson. Twisted Edwards curves revisited. In J. Pieprzyk, editor, *Advances in Cryptology - ASIACRYPT 2008*, volume 5350 of *Lecture Notes in Computer Science*, pages 326–343. Springer, 2008.
34. M. Hutter and P. Schwabe. Multiprecision multiplication on AVR revisited. *Journal of Cryptographic Engineering*, 5(3):201–214, 2015.
35. T. Izu and T. Takagi. Exceptional procedure attack on elliptic curve cryptosystems. In Y. Desmedt, editor, *Public Key Cryptography - PKC 2003*, volume 2567 of *Lecture Notes in Computer Science*, pages 224–239. Springer, 2003.
36. K. Järvinen, A. Miele, R. Azarderakhsh, and P. Longa. Four$\mathbb{Q}$ on FPGA: new hardware speed records for elliptic curve cryptography over large prime characteristic fields. In B. Gierlichs and A. Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016*, volume 9813 of *Lecture Notes in Computer Science*, pages 517–537. Springer, 2016.
37. M. Joye and C. Tymen. Protections against differential analysis for elliptic curve cryptography. In *Cryptographic Hardware and Embedded Systems - CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 377–390. Springer, 2001.
38. M. Joye and S. Yen. The Montgomery powering ladder. In B. S. K. Jr., Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 291–302. Springer, 2002.
39. P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. J. Wiener, editor, *Advances in Cryptology - CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.
40. P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *Advances in Cryptology - CRYPTO'96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.
41. B. Krebs. Hacked cameras, DVRs powered todays massive Internet outage, October 2016. `https://krebsonsecurity.com/2016/10/hacked-cameras-dvrs-powered-todays-massive-internet-outage/`.
42. W. Ladd, P. Longa, and R. Barnes. Curve4Q. *Internet-Draft, draft-ladd-cfrg-4q-01*, 2016-2017. Available at: `https://www.ietf.org/id/draft-ladd-cfrg-4q-01.txt`.
43. Z. Liu, H. Seo, Z. Hu, X. Hunag, and J. Großschädl. Efficient implementation of ECDH key exchange for MSP430-based wireless sensor networks. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 145–153. ACM, 2015.
44. I. Liusvaara and S. Josefsson. Edwards-curve Digital Signature Algorithm (EdDSA). Internet-Draft draft-irtf-cfrg-eddsa-05, Internet Engineering Task Force, 2016. Work in Progress. Available at: `https://tools.ietf.org/html/draft-irtf-cfrg-eddsa-05`.
45. P. Longa. Four$\mathbb{Q}$NEON: faster elliptic curve scalar multiplications on ARM processors. In R. Avanzi and H. Heys, editors, *Selected Areas in Cryptography - SAC 2016 (to appear)*, Lecture Notes in Computer Science. Springer, 2016. Available at: `http://eprint.iacr.org/2016/645`.
46. H. Mamiya, A. Miyaji, and H. Morimoto. Efficient countermeasures against RPA, DPA, and SPA. In *Cryptographic Hardware and Embedded Systems - CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 343–356. Springer, 2004.
47. R. P. McEvoy, M. Tunstall, C. C. Murphy, and W. P. Marnane. Differential power analysis of HMAC based on SHA-2, and countermeasures. In S. Kim, M. Yung, and H. Lee, editors, *Information Security Applications, WISA 2007*, volume 4867 of *Lecture Notes in Computer Science*, pages 317–332. Springer, 2008.
48. C. Murdica, S. Guilley, J. Danger, P. Hoogvorst, and D. Naccache. Same values power analysis using special points on elliptic curves. In W. Schindler and S. A. Huss, editors, *Constructive Side-Channel*

Analysis and Secure Design - COSADE 2012, volume 7275 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2012.
49. E. Nascimento, L. Chmielewski, D. Oswald, and P. Schwabe. Attacking embedded ECC implementations through cmov side channels. *Selected Areas in Cryptology – SAC 2016, Springer-Verlag (to appear)*, 2016.
50. E. Nascimento, J. López, and R. Dahab. Efficient and secure elliptic curve cryptography for 8-bit AVR microcontrollers. In R. S. Chakraborty, P. Schwabe, and J. A. Solworth, editors, *Security, Privacy, and Applied Cryptography Engineering - SPACE 2015*, volume 9354 of *Lecture Notes in Computer Science*, pages 289–309. Springer, 2015.
51. K. Okeya and K. Sakurai. Power analysis breaks elliptic curve cryptosystems even secure against the timing attack. In B. K. Roy and E. Okamoto, editors, *Progress in Cryptology - INDOCRYPT 2000*, volume 1977 of *Lecture Notes in Computer Science*, pages 178–190. Springer, 2000.
52. E. Oswald and S. Mangard. Template attacks on masking - resistance is futile. In M. Abe, editor, *Topics in Cryptology - CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007*, volume 4377 of *Lecture Notes in Computer Science*, pages 243–256. Springer, 2007.
53. D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. Technical report CSTR-02-003, Department of Computer Science, University of Bristol, 2002. `http://www.cs.bris.ac.uk/Publications/Papers/1000625.pdf`.
54. K. Piotrowski, P. Langendoerfer, and S. Peter. How public key cryptography influences wireless sensor node lifetime. In *Proceedings of the fourth ACM workshop on Security of ad hoc and sensor networks*, pages 169–176. ACM, 2006.
55. E. Prouff and M. Rivain. A generic method for secure sbox implementation. In S. Kim, M. Yung, and H. Lee, editors, *Information Security Applications, WISA 2007*, volume 4867 of *Lecture Notes in Computer Science*, pages 227–244. Springer, 2007.
56. J. Renes, P. Schwabe, B. Smith, and L. Batina. μKummer: Efficient hyperelliptic signatures and key exchange on microcontrollers. In B. Gierlichs and A. Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016*, volume 9813 of *Lecture Notes in Computer Science*, pages 301–320. Springer, 2016.
57. O. Reparaz, B. Gierlichs, and I. Verbauwhede. Selecting time samples for multivariate DPA attacks. In E. Prouff and P. Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 155–174. Springer, 2012.
58. E. Ronen and A. Shamir. Extended functionality attacks on IoT devices: The case of smart lights. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pages 3–12. IEEE, 2016.
59. F. D. Santis and G. Sigl. Towards side-channel protected X25519 on ARM Cortex-M4 processors. *Software performance enhancement for encryption and decryption, and benchmarking (SPEED-B)*, 2016.
60. C. Schnorr. Efficient identification and signatures for smart cards. In G. Brassard, editor, *Advances in Cryptology - CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 239–252. Springer, 1990.
61. H. Seo, Z. Liu, J. Choi, and H. Kim. Multi-precision squaring for public-key cryptography on embedded microprocessors. In *International Conference on Cryptology in India*, pages 227–243. Springer, 2013.
62. H. Seuschek, J. Heyszl, and F. D. Santis. A cautionary note: Side-channel leakage implications of deterministic signature schemes. In M. Palkovic, G. Agosta, A. Barenghi, I. Koren, and G. Pelosi, editors, *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems, CS2@HiPEAC, Prague, Czech Republic, January 20, 2016*, pages 7–12. ACM, 2016.
63. N. P. Smart, E. Oswald, and D. Page. Randomised representations. *IET Information Security*, 2(2):19–27, 2008.
64. B. Smith. The $\mathbb{Q}$-curve construction for endomorphism-accelerated elliptic curves. *J. Cryptology*, 29(4):806–832, 2016.
65. F. Standaert, N. Veyrat-Charvillon, E. Oswald, B. Gierlichs, M. Medwed, M. Kasper, and S. Mangard. The world is not enough: Another look on second-order DPA. In M. Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, volume 6477 of *Lecture Notes in Computer Science*, pages 112–129. Springer, 2010.
66. STMicroelectronics. STM32F405xx/STM32F407xx, datasheet, 2016. `http://www.st.com/content/ccc/resource/technical/document/datasheet/ef/92/76/6d/bb/c2/4f/f7/DM00037051.pdf/files/DM00037051.pdf/jcr:content/translations/en.DM00037051.pdf`.
67. STMicroelectronics. STM32F4DISCOVERY: Discovery kit with STM32F407VG MCU, data brief, 2016. `http://www.st.com/content/ccc/resource/technical/document/data_brief/09/71/8c/4e/e4/da/4b/fa/DM00037955.pdf/files/DM00037955.pdf/jcr:content/translations/en.DM00037955.pdf`.

68. STMicroelectronics. Reference manual: STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced ARM-based 32-bit MCUs, 2017. http://www.st.com/content/ccc/resource/technical/document/reference_manual/3d/6d/5a/66/b4/99/40/d4/DM00031020.pdf/files/DM00031020.pdf/jcr:content/translations/en.DM00031020.pdf.
69. Texas Instruments. User's guide: MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx family, 2012–2017. http://www.ti.com.cn/cn/lit/ug/slau367m/slau367m.pdf.
70. M. Tunstall, C. Whitnall, and E. Oswald. Masking tables - an underestimated security risk. In S. Moriai, editor, *Fast Software Encryption - FSE 2013*, volume 8424 of *Lecture Notes in Computer Science*, pages 425–444. Springer, 2014.
71. E. Wenger, T. Unterluggauer, and M. Werner. 8/16/32 shades of elliptic curve cryptography on embedded processors. In G. Paul and S. Vaudenay, editors, *Progress in Cryptology - INDOCRYPT 2013*, volume 8250 of *Lecture Notes in Computer Science*, pages 244–261. Springer, 2013.

A Appendix: scalar decomposition and recoding routines

The decomposition and multiscalar recoding routines, which are required by Algorithm 1 at Steps 3 and 4 and Algorithm 2 at Step 6, are presented in Listings 1.1 and 1.2, respectively. The decomposition pseudocode receives as input a 256-bit scalar written as an array with the form `k = (k[0],k[1],k[2],k[3])` consisting of four 64-bit unsigned digits in little-endian form, and outputs the decomposed multiscalar array `a = (a[0],a[1],a[2],a[3])` consisting of four 64-bit unsigned digits. The function `mul_truncate(a,b,c)` included in the decomposition algorithm performs the computation `c = (uint64_t)((a * b) >> 256)`, and the required constants are listed in Listing 1.3. Likewise, the recoding pseudocode receives as input `a`, produced by the decomposition routine, and outputs the 65-digit array `d = (d[0], ..., d[64])` and the 65-mask array `m = (m[0], ..., m[64])`. Note that the datatype `uint64_t` is written as `u64_t` in the pseudocodes.

```
void decompose(const u64_t k[],u64_t a[]) {
 u64_t a1,a2,a3,a4,t,m;

 mul_truncate(k,ell1,&a1);
 mul_truncate(k,ell2,&a2);
 mul_truncate(k,ell3,&a3);
 mul_truncate(k,ell4,&a4);

 t = k[0] - (u64_t)a1*b11 - (u64_t)a2*b21 - (u64_t)a3*b31 - (u64_t)a4*b41 + c1;
 m = ~(0 - (t & 1)); // If t is even then m = 0xFF...FF, else m = 0

 a[0] = t + (m & b41);
 a[1] = (u64_t)a1*b12 + (u64_t)a2     - (u64_t)a3*b32 - (u64_t)a4*b42 + c2 + (m & b42);
 a[2] = (u64_t)a3*b33 - (u64_t)a1*b13 - (u64_t)a2     + (u64_t)a4*b43 + c3 - (m & b43);
 a[3] = (u64_t)a1*b14 - (u64_t)a2*b24 - (u64_t)a3*b34 + (u64_t)a4*b44 + c4 - (m & b44);
}
```

Listing 1.1. Scalar decomposition routine.

```
void recode(const u64_t a[],uint d[],uint m[]) {
 uint i, bit,bit0,carry;
 m[64] = (uint)-1;

 for (i = 0; i < 64; i++) {
     a[0] >>= 1;
     bit0 = (uint)a[0] & 1;
     m[i] = 0 - bit0;

     bit = (uint)a[1] & 1;
     carry = (bit0 | bit) ^ bit0;
```

```
    a[1] = (a[1] >> 1) + (u64_t)carry;
    d[i] = bit;

    bit = (uint)a[2] & 1;
    carry = (bit0 | bit) ^ bit0;
    a[2] = (a[2] >> 1) + (u64_t)carry;
    d[i] += (bit << 1);

    bit = (uint)a[3] & 1;
    carry = (bit0 | bit) ^ bit0;
    a[3] = (a[3] >> 1) + (u64_t)carry;
    d[i] += (bit << 2);
 }
 d[64] = (uint)(a[1] + (a[2] << 1) + (a[3] << 2));
}
```

Listing 1.2. Multiscalar recoding routine.

```
// Close "offset" vector:
 u64_t c1  = {0x72482C5251A4559C};
 u64_t c2  = {0x59F95B0ADD276F6C};
 u64_t c3  = {0x7DD2D17C4625FA78};
 u64_t c4  = {0x6BC57DEF56CE8877};
// Optimal basis vectors:
 u64_t b11 = {0x0906FF27E0A0A196};
 u64_t b12 = {0x1363E862C22A2DA0};
 u64_t b13 = {0x07426031ECC8030F};
 u64_t b14 = {0x084F739986B9E651};
 u64_t b21 = {0x1D495BEA84FCC2D4};
 u64_t b24 = {0x25DBC5BC8DD167D0};
 u64_t b31 = {0x17ABAD1D231F0302};
 u64_t b32 = {0x02C4211AE388DA51};
 u64_t b33 = {0x2E4D21C98927C49F};
 u64_t b34 = {0x0A9E6F44C02ECD97};
 u64_t b41 = {0x136E340A9108C83F};
 u64_t b42 = {0x3122DF2DC3E0FF32};
 u64_t b43 = {0x068A49F02AA8A9B5};
 u64_t b44 = {0x18D5087896DE0AEA};
// Precomputed integers for fast-Babai rounding (in little-endian form):
 u64_t ell1[4] = {0x259686E09D1A7D4F, 0xF75682ACE6A6BD66, 0xFC5BB5C5EA2BE5DF, 0x07};
 u64_t ell2[4] = {0xD1BA1D84DD627AFB, 0x2BD235580F468D8D, 0x8FD4B04CAA6C0F8A, 0x03};
 u64_t ell3[4] = {0x9B291A33678C203C, 0xC42BD6C965DCA902, 0xD038BF8D0BFFBAF6, 0x00};
 u64_t ell4[4] = {0x12E5666B77E7FDC0, 0x81CBDC3714983D82, 0x1B073877A22D8410, 0x03};
```

Listing 1.3. Constants for the decomposition routine.