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

Zhe Liu[1,2], Patrick Longa[3], Geovandro C. C. F. Pereira[2], Oscar Reparaz[4], and Hwajeong Seo[5]

[1] SnT, University of Luxembourg, Luxembourg
[2] IQC, University of Waterloo, Canada
{zhelu.liu,geovandro.pereira}@uwaterloo.ca
[3] Microsoft Research, USA
plonga@microsoft.com
[4] imec-COSIC KU Leuven, Belgium
oscar.reparaz@esat.kuleuven.be
[5] Hansung University, Korea
hwajeong84@gmail.com

Abstract. This work deals with the energy-efficient, high-speed and high-security implementation of elliptic curve scalar multiplication and elliptic curve Diffie-Hellman (ECDH) key exchange 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 and DH key exchange 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 ∼6.9 million cycles (or 0.86 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 ∼496 thousand cycles on a 32-bit ARM Cortex-M4 microcontroller, achieving a factor-2.9 speedup when compared to the fastest Curve25519 implementation targeting the same platform. 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. 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 IoT.

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

1 Introduction

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.

Four$\mathbb{Q}$ [16] 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 [16, 36]. 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.

Side-channel attacks. Protection against side-channel attacks [33, 32] represents another important aspect of the security in embedded devices. These attacks, which have been the focus of intense research since Kocher' seminal paper [33], can be classified as: *passive* attacks (a.k.a. side-channel analysis (SCA)), such as differential side-channel analysis (DSCA) [32], timing [33], correlation [6], collision [23] and template [8] attacks, among many other variants; and *active* attacks (a.k.a. fault attacks). Refer to [3, 20] 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 and ECDH key exchange 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 2.9x faster than the fastest Curve25519 implementations on 8-bit AVR, 16-bit MSP430X, and 32-bit ARM Cortex-M4 MCUs, respectively.

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 and ECDH algorithms, which include a set of efficient countermeasures that have been especially tailored for Four$\mathbb{Q}$, are designed to min-

[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.

imize 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 [23], the refined power attack (RPA) [25], zero-value point attacks (ZVP) [1], same value attacks (SVA) [38], exceptional procedure attacks [29], 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 [55, 40]. Moreover, reported implementations (other than implementations exclusively protected against timing attacks [19]) have not been publicly released. Our software for ARM Cortex-M4 has been made publicly available as part of the Fourℚlib library [17]:

`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 can 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 [42]. 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.

2 Preliminaries: Fourℚ

Fourℚ, introduced by Costello and Longa in 2015 [16], 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 $d = 125317048443780598345676279555970305165 \cdot i + 4205857648805777768770$. 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 $\mathcal{O}_{\mathcal{E}} = (0, 1)$.

Fourℚ 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 $\mathcal{E}(\mathbb{F}_{p^2})[N]$, proceeds

Algorithm 1 Four$\mathbb{Q}$'s scalar multiplication on $\mathcal{E}(\mathbb{F}_{p^2})[N]$ (from [16]).

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 [16, Prop. 5].
4: Recode (a_1, a_2, a_3, a_4) into $(d_{64}, \ldots, d_0)$ and $(m_{64}, \ldots, m_0)$ using [16, Alg. 1]. Write $s_i = 1$ if $m_i = -1$ and $s_i = -1$ if $m_i = 0$.

Main loop:

5: $Q = s_{64} \cdot T[d_{64}]$
6: **for** $i = 63$ **to** 0 **do**
7: $\quad Q = [2]Q + s_i \cdot T[d_i]$
8: **return** Q

as follows (see Algorithm 1). First, one needs to prepare an 8-point precomputed table at Steps 1–2 and execute the decomposition and recoding algorithms at Steps 3–4. As before, let a scalar m be any integer in $[1, 2^{256})$. Four$\mathbb{Q}$'s decomposition [16, Prop. 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 [16, Alg. 1] to a representation consisting of exactly 65 "signed digit-columns" d_j and "sign masks" m_j for $j = 0, ..., 64$. Finally, the evaluation stage consists of an initial point loading and a single loop of 64 iterations, where each iteration consists of exactly one doubling and one addition.

2.1 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" [34]:

function DH(m, P)
  Check that P is on the curve. If not, return "FAILED".
  Compute $Q = [392]P$ and $T = [m]Q$.
  If $T = (0, 1)$ return "FAILED".
Return T in affine coordinates.

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.

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 =DH(m_A, B) and Bob computes K_B =DH(m_B, A). The y-coordinate of the value $K = K_A = K_B$ can then be used as the shared secret.

ECDH key exchange with 32-byte public keys. It is possible to reduce the size of the public keys to only 32 bytes with the approach described next.

An element $y = a + b \cdot i \in \mathbb{F}_{p^2}$ encoded as $\underline{y} = (a_0, ..., a_{126}, 0, b_0, ..., b_{126})$ is defined as "negative" if and only if $a_{126} = 1$, or if $b_{126} = 1$ and $a = 0$. We define Compress(P) as the function that takes as input a point $P = (x, y) \in E$ and encodes it 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. We define Expand(S) as the function that takes a 256-bit string S and 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 S. Refer to [18, Appendix A] and [34, Appendix B] for low-level details about the decompression procedure.

The ECDH key exchange mechanism then proceeds as follows [34]. Alice and Bob pick random integers m_A and m_B (resp.) in the range $[0, 2^{256})$, and then compute the public keys $\underline{A}$ = Compress$([m_A]G)$ and $\underline{B}$ = Compress$([m_B]G)$ (resp.). After exchanging public keys, Alice computes K_A = DH$(m_A$, Expand$(\underline{B}))$ and Bob computes K_B = DH$(m_B$, Expand$(\underline{A}))$. As before, the y-coordinate of the value $K = K_A = K_B$ is the shared secret.

3 Implementation details on AVR, MSP and ARM

In this section, we briefly describe relevant implementation aspects for *three* popular MCUs: 8-bit AVR ATmega, 16-bit MSP430X, and 32-bit ARM Cortex-M4. For more details, refer to the extended version of the paper [35].

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

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

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 $r \leftarrow r \pmod{2^{127}} + (r \gg 127)$.

3.2 Implementation on 8-bit AVR ATmega

AVR microcontrollers have 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.

Finite field operations. For the 128-bit integer multiplication, we use 2-level Karatsuba in a recursive way, with the low level 32-bit multiplications implemented in product-scanning form. For the 128-bit squaring, we employ the Sliding Block Doubling (SBD) method [49]. In order to minimize the use of load/store instructions, integer multiplication and squaring were integrated with the modular reduction at the assembly level. Modular reduction over the prime $p = 2^{127} - 1$ as well as the arithmetic over $\mathbb{F}_{p^2}$ were implemented as described in §3.1.

3.3 Implementation on 16-bit MSP430X

The ultra-low power MSP430X is a representative 16-bit microcontroller 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 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.

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 [26]. Squaring was implemented using the SBD method, as in the case of AVR. Modular reduction and the arithmetic over $\mathbb{F}_{p^2}$ were implemented as described in §3.1.

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. 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. 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 [52] that contains a 32-bit ARM Cortex-M4F STM32F407VGT6 microcontroller. 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. Integer multiplication was implemented using the schoolbook method and the efficient MAC instructions. The computation of a field multiplication is then completed with the execution of 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 $a = a_0 + a_1 \cdot i, b = b_0 + b_1 \cdot i \in \mathbb{F}_{p^2}$.

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. 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 [21, Alg. 5], which requires a table with $v \cdot 2^{w-1}$ points (v and w denote the number of internal tables and their window size, respectively).

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 ATmega MCU, since its architecture does not support the use of cache memory. Although the MSP430FR MCU family presents some form of integrated caching, it is activated when the MCU operates at a higher frequency than the access frequency of the FRAM [54] (i.e., 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. Finally, the targeted Cortex-M4 STM32F4 MCU includes a cache memory to accelerate flash memory accesses [53]. However, our software does not use flash memory to store the precomputed tables and, therefore, cache attacks do not apply.

At the high-level, we implemented the ECDH schemes described in §2.1, which are protected against invalid point and small subgroup attacks.

Results. Table 1 summarizes the results for variable-base and fixed-base scalar multiplication, static ECDH and fully ephemeral ECDH key exchange for the three targeted microcontrollers. In the case of ECDH with Four$\mathbb{Q}$, we evaluate the use of both 32 and 64-byte public keys. For comparison, we include two efficient alternatives that have been deployed on various microcontrollers: the "μKummer" implementation by Renes et al. [46] using the genus-2 Kummer surface by Gaudry and Schost [24], and the "Curve25519" implementations by Düll et al. [19] and De Santis et al. [48]. 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 [19] and [48]).

As can be seen in Table 1, 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,

Table 1. 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. Cycle counts are rounded to the nearest 10^2 cycles.

Source	**scalar multiplication**		**ECDH**	
	fixed-base	**random**	**static**	**ephemeral**
8-bit AVR ATmega				
Curve25519 [19]	13,900,400[1]	13,900,400	13,900,400[3]	27,800,800[2,3]
μKummer [46]	9,513,500[1]	9,513,500	9,739,100[4]	19,027,100[2,4]
Four$\mathbb{Q}$ (this work)	**2,980,700**	**6,505,300**	**6,886,400**[5] **7,221,300**[3]	**9,870,500**[5] **10,206,500**[3]
16-bit MSP430X (16-bit multiplier) @8MHz				
Curve25519 [19]	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 [48]	1,423,700[1]	1,423,700	1,423,700[3]	2,847,400[2,3]
Four$\mathbb{Q}$ (this work)	**232,900**	**469,500**	**496,400**[5] **542,900**[3]	**729,900**[5] **776,600**[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, respectively.

1.9x, and 3x 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 3.9x 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.

As consequence of the reduction in computing time, our implementations allow a significant reduction in energy costs. For example, following [44] we estimate that our software demands 41.65mJ of energy to compute a fully ephemeral ECDH key exchange with 32-byte public keys (or, equivalently, $\sim 162,064$ key exchanges for the life of a double AA battery) on a MICAz sensor node containing an 8-bit AVR MCU. When comparing against similar calculations for other curves, we observe that our Four$\mathbb{Q}$ implementation on AVR is able to run 2.7x and 1.9x more key exchanges than Curve25519 and μKummer (resp.) for the same battery budget.

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 and cover implementation aspects of table lookups and a protected ECDH key exchange scheme. 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 [47]). 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 [15]. Other popular methods include key splitting [10], and random curve and field isomorphisms [30].

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 [41, 9, 10, 50]. 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. [11] 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 [16, 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 [16, 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}*.*

Refer to the extended version of the paper [35] for the proof of Proposition 1.

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 FourQ's scalar multiplication is only increased by 25% in this case).

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. [37]. 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 $[(1\bar{1}\bar{1}\dots\bar{1})_2]R$.

Adapting EBRIP to FourQ is straightforward: it suffices to assume $t = 4$ and adjust the precomputed values which, in the case of FourQ, 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 [41, 23]. 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.

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 [15]. 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 §2.1 only needs minor changes and the inclusion of a blinding point B. We assume that a fresh blinding point is generated during key generation. The modified function is shown below.

function DH_SCA(m, P, B)
  Check that P and B are on the curve. If not, return "FAILED".
  Compute $Q = [392]P$.
  Compute $T = [m]Q$ and update B using Algorithm 2.
  If $T = (0, 1)$ return "FAILED", else return T and B in affine.

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

Algorithm 2 SCA-protected FourQ'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 $0 \le u \le 15$. 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 [16, Prop. 5].

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: $S = (r_i \cdot X_{T[d_i]}, r_i \cdot Y_{T[d_i]}, r_i \cdot Z_{T[d_i]})$.

11: $Q = [2]Q + S$

12: **return** $(Q - R)$ and R in affine coordinates.

described in §2.1. As explained before, these functions are protected against invalid point and small subgroup attacks.

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.

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 [28]. 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 exceptional procedure attacks [29]. Likewise, [16, 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 [33] and cache attacks [43]. 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 [32]. 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 [25], ZVP [1] and SVA [38] 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 [23]. We first note that intermediate values in the EBRIP algorithm [37] 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 a 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 FourQ offer better protection in comparison with other elliptic curves. For example, Feix et al. [22] 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 [27]). 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 [40]). However, other more sophisticated countermeasures might be required to protect against recent one-trace template attacks that inspect memory accesses [39]. 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 1.18 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 1). 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 2.60 and 2.55 million cycles, resp.

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. [12] 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. [31, §4.1] or [45, §3.1]) and is usually provided by the underlying hardware.

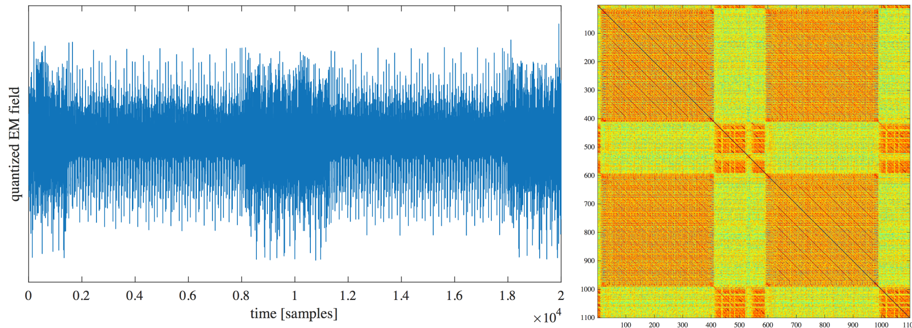

Fig. 1. Left: exemplary EM trace. Right: cross correlation of a single trace.

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, 51].

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 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 test [14, 13] 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.

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

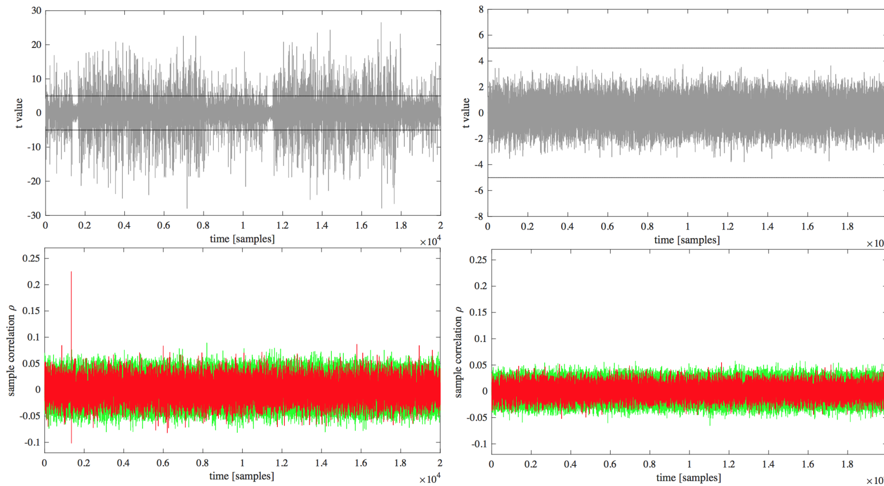

Fig. 2. Top row: fixed-vs-random leakage detection test on the input point. Bottom: CPA attacks. Left column: no countermeasure enabled. Right column: point blinding on/coord. randomization off/scalar randomization off.

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.)

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

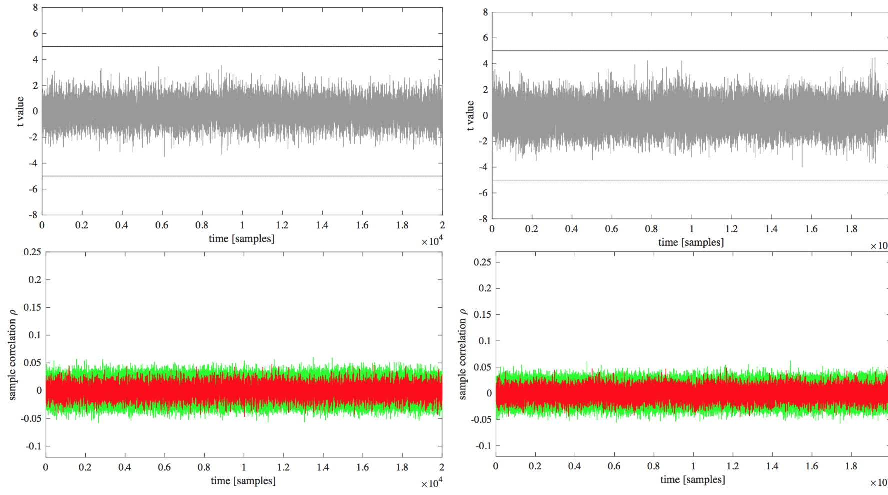

Fig. 3. Left: point blinding off/coord. randomization on/scalar randomization off. Right: point blinding off/coord. randomization off/scalar randomization on.

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.

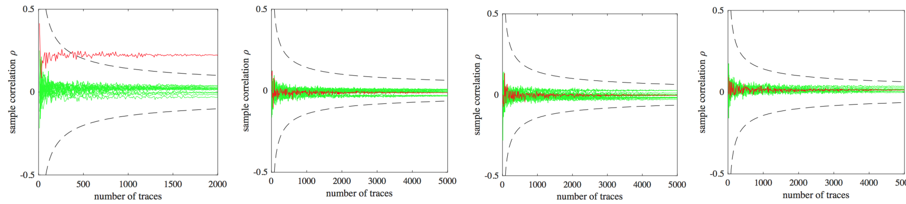

Fig. 4. Evolution of ρ as function of number of traces. Left to right (point blinding/coord. randomization/scalar randomization): off/off/off, on/off/off, off/on/off, off/off/on.

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 Acknowledgments

We would like to thank Craig Costello for helping in the design of the scalar randomization countermeasure, and Diego F. Aranha, Pedro R. N. Pedruzzi, Joost Renes and the reviewers for their valuable comments. Geovandro Pereira was partially supported by NSERC, CryptoWorks21, and Public Works and Government Services Canada. Oscar Reparaz was partially supported by the Research Council KU Leuven C16/15/058. Hwajeong Seo was supported by the ICT R&D program of MSIP/IITP (B0717-16-0097, Development of V2X Service Integrated Security Technology for Autonomous Driving Vehicle).

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 *LNCS*, 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. http://eprint.iacr.org/2005/017.
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 *LNCS*, 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 *LNCS*, 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 *LNCS*, 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 *LNCS*, 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 *LNCS*, pages 13–28. Springer, 2002.
9. M. Ciet. *Aspects of fast and secure arithmetics for elliptic curve cryptography.* PhD thesis, Université Catholique de Louvain, Louvain-la-Neuve, 2003.
10. 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 *LNCS*, pages 348–359. Springer, 2003.
11. 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 *LNCS*, pages 540–550. Springer, 2002.

12. 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 *LNCS*, pages 46–61. Springer, 2010.
13. 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.
14. J. Coron, P. C. Kocher, and D. Naccache. Statistics and secret leakage. In Y. Frankel, editor, *Financial Cryptography, FC 2000*, volume 1962 of *LNCS*, pages 157–173. Springer, 2000.
15. 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 *LNCS*, pages 292–302. Springer, 1999.
16. 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 *LNCS*, pages 214–235. Springer, 2015. Full version: `https://eprint.iacr.org/2015/565`.
17. C. Costello and P. Longa. Four$\mathbb{Q}$lib. `https://github.com/Microsoft/FourQlib`, 2015-2017.
18. 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`.
19. 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.
20. 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 *LNCS*, pages 265–282. Springer, 2012.
21. 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.
22. 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 *LNCS*, pages 3–20. Springer, 2014.
23. 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 *LNCS*, pages 269–280. Springer, 2003.
24. P. Gaudry and E. Schost. Genus 2 point counting over prime fields. *J. Symbolic Computation*, 47(4):368–400, 2012.
25. L. Goubin. A refined power-analysis attack on elliptic curve cryptosystems. In Y. Desmedt, editor, *Public Key Cryptography - PKC 2003*, volume 2567 of *LNCS*, pages 199–210. Springer, 2003.
26. C. P. L. Gouvêa and J. López. Software implementation of pairing-based cryptography on sensor networks using the MSP430 microcontroller. In B. K. Roy and N. Sendrier, editors, *Progress in Cryptology - INDOCRYPT 2009*, pages 248–262. Springer, 2009.

27. 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 *LNCS*, pages 431–448. Springer, 2015.
28. 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 *LNCS*, pages 326–343. Springer, 2008.
29. T. Izu and T. Takagi. Exceptional procedure attack on elliptic curve cryptosystems. In Y. Desmedt, editor, *Public Key Cryptography - PKC 2003*, volume 2567 of *LNCS*, pages 224–239. Springer, 2003.
30. M. Joye and C. Tymen. Protections against differential analysis for elliptic curve cryptography. In *Cryptographic Hardware and Embedded Systems - CHES 2001*, volume 2162 of *LNCS*, pages 377–390. Springer, 2001.
31. 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 *LNCS*, pages 291–302. Springer, 2002.
32. P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. J. Wiener, editor, *Advances in Cryptology - CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.
33. 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 *LNCS*, pages 104–113. Springer, 1996.
34. 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`.
35. Z. Liu, P. Longa, G. Pereira, O. Reparaz, and H. Seo. Four$\mathbb{Q}$ on embedded devices with strong countermeasures against side-channel attacks. *IACR Cryptology ePrint Archive, Report 2017/434*, 2017. `http://eprint.iacr.org/2017/434`.
36. 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)*, LNCS. Springer, 2016. Available at: `http://eprint.iacr.org/2016/645`.
37. 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 *LNCS*, pages 343–356. Springer, 2004.
38. 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 *LNCS*, pages 183–198. Springer, 2012.
39. 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.
40. 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 *LNCS*, pages 289–309. Springer, 2015.
41. 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 *LNCS*, pages 178–190. Springer, 2000.

42. 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 *LNCS*, pages 243–256. Springer, 2007.
43. 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.
44. 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.
45. 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 *LNCS*, pages 227–244. Springer, 2007.
46. 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 *LNCS*, pages 301–320. Springer, 2016.
47. 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 *LNCS*, pages 155–174. Springer, 2012.
48. 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.
49. 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.
50. N. P. Smart, E. Oswald, and D. Page. Randomised representations. *IET Information Security*, 2(2):19–27, 2008.
51. 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 *LNCS*, pages 112–129. Springer, 2010.
52. 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.
53. 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.
54. Texas Instruments. User's guide: MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx family, 2012–2017. http://www.ti.com.cn/cn/lit/ug/slau367m/slau367m.pdf.
55. 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 *LNCS*, pages 244–261. Springer, 2013.