FourQ-based cryptography for high-performance and low-power applications

Real World Cryptography Conference 2017
January 4-6, New York, USA

Patrick Longa
Microsoft Research

Microsoft Research

Next-generation elliptic curves

New IETF Standards

- The Crypto Forum Research Group (CFRG) selected two elliptic curves: Bernstein's Curve25519 and Hamburg's Ed448-Goldilocks
- RFC 7748: "Elliptic Curves for Security" (published on January 2016)
  - Curve details; generation
  - DH key exchange for both curves
- Ongoing work: signature scheme
  - draft-irtf-cfrg-eddsa-08, "Edwards-curve Digital Signature Algorithm (EdDSA)"

Next-generation elliptic curves

Farrel-Moriarity-Melkinov-Paterson [NIST ECC Workshop 2015]:

"… the real motivation for work in CFRG is the better performance and side-channel resistance of new curves developed by academic cryptographers over the last decade."

Plus some additional requirements such as:

- Rigidity in curve generation process.
- Support for existing cryptographic algorithms.

Next-generation elliptic curves

Farrel-Moriarity-Melkinov-Paterson [NIST ECC Workshop 2015]:

"… the real motivation for work in CFRG is the better performance and side-channel resistance of new curves developed by academic cryptographers over the last decade."

Plus some additional requirements such as:

- Rigidity in curve generation process.
- Support for existing cryptographic algorithms.

State-of-the-art ECC: Four$\mathbb{Q}$

[Costello-L, ASIACRYPT 2015]

- CM endomorphism [GLV01] and Frobenius ($\mathbb{Q}$-curve) endomorphism [GLS09, Smi16, GI13]
- Edwards form [Edw07] using efficient Edwards coordinates [BBJ+08, HCW+08]
- Arithmetic over the Mersenne prime $p = 2^{127} - 1$

Four$\mathbb{Q}$

Features:

- Support for secure implementations and top performance.
- Uniqueness: only curve at the 128-bit security level with properties above.

State-of-the-art ECC: Four$\mathbb{Q}$
[Costello-L, ASIACRYPT 2015]

- CM endomorphism [GLV01] and Frobenius ($\mathbb{Q}$-curve) endomorphism [GLS09, Smi16, GI13]
- Edwards form [Edw07] using efficient Edwards coordinates [BBJ+08, HCW+08]
- Arithmetic over the Mersenne prime $p = 2^{127} - 1$

Four$\mathbb{Q}$

Features:
- Support for secure implementations and top performance.
- Uniqueness: only curve at the 128-bit security level with properties above.

State-of-the-art ECC: Four$\mathbb{Q}$

[Costello-L, ASIACRYPT 2015]

Speed (in thousands of cycles) to compute variable-base scalar multiplication on different computer classes.

Platform	Four$\mathbb{Q}$	Curve25519	Speedup ratio
Intel Haswell processor, **desktop class**	**56**	162	**2.9x**
ARM Cortex-A15, **smartphone class**	**132**	315	**2.4x**
ARM Cortex-M4, **microcontroller class**	**531**	1,424	**2.7x**

State-of-the-art ECC: Four$\mathbb{Q}$
[Costello-L, ASIACRYPT 2015]

Speed (in thousands of cycles) to compute variable-base scalar multiplication on different computer classes.

Platform	Four$\mathbb{Q}$	Curve25519	Speedup ratio
Intel Haswell processor, **desktop class**	**56**	162	**2.9x**
ARM Cortex-A15, **smartphone class**	**132**	315	**2.4x**
ARM Cortex-M4, **microcontroller class**	**531**	1,424	**2.7x**

State-of-the-art ECC: Four$\mathbb{Q}$
[Costello-L, ASIACRYPT 2015]

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

$d = 125317048443780598345676279555970305165i + 4205857648805777768770,$
$p = 2^{127} - 1$, $i^2 = -1$, $\#E = 392 \cdot N$, where N is a 246-bit prime.

State-of-the-art ECC: Four$\mathbb{Q}$
(Costello-L, ASIACRYPT 2015)

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

$$d = 125317048443780598345676279555970305165i + 4205857648805777768770,$$
$p = 2^{127} - 1$, $i^2 = -1$, $\#E = 392 \cdot N$, where N is a 246-bit prime.

- Fastest (large char) ECC addition laws **are complete** on E
- E is equipped with *two* endomorphisms:
  - E is a degree-2 $\mathbb{Q}$-curve: endomorphism ψ
  - E has CM by order of $D = -40$: endomorphism ϕ

State-of-the-art ECC: Four$\mathbb{Q}$
(Costello-L, ASIACRYPT 2015)

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

$$d = 125317048443780598345676279555970305165i + 4205857648805777768770,$$
$p = 2^{127} - 1$, $i^2 = -1$, $\#E = 392 \cdot N$, where N is a 246-bit prime.

- Fastest (large char) ECC addition laws **are complete** on E
- E is equipped with *two* endomorphisms:
  - E is a degree-2 $\mathbb{Q}$-curve: endomorphism ψ
  - E has CM by order of $D = -40$: endomorphism ϕ
- $\psi(P) = [\lambda_\psi]P$ and $\phi(P) = [\lambda_\phi]P$ for all $P \in E[N]$ and $m \in [0, 2^{256})$

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

Optimal 4-Way Scalar Decompositions

$$m \mapsto (a_1, a_2, a_3, a_4)$$

Proposition: for all $m \in [0, 2^{256}\rangle$, decomposition yields four $a_i \in [0, 2^{64}\rangle$ with a_1 odd.

$$m = 42453556751700041597675664513313229052985088397396902723728803518727612539248$$

$$\begin{aligned} a_1 &= 13045455764875651153 && P \\ a_2 &= 9751504369311420685 && \phi(P) \\ a_3 &= 5603607414148260372 && \psi(P) \\ a_4 &= 8360175734463666813 && \psi(\phi(P)) \end{aligned}$$

Optimal 4-Way Scalar Decompositions

$$m \mapsto (a_1, a_2, a_3, a_4)$$

Proposition: for all $m \in [0, 2^{256}\rangle$, decomposition yields four $a_i \in [0, 2^{64}\rangle$ with a_1 odd.

$$m = 42453556751700041597675664513313229052985088397396902723728803518727612539248$$

$$\begin{aligned} a_1 &= 13045455764875651153 && P \\ a_2 &= 9751504369311420685 && \phi(P) \\ a_3 &= 5603607414148260372 && \psi(P) \\ a_4 &= 8360175734463666813 && \psi(\phi(P)) \end{aligned}$$

Multi-Scalar Recoding

Step 1: recode a_1 to signed non-zero representation

Step 2: recode a_2, a_3 and a_4 by “sign-aligning” columns

$$a_1 = 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1$$
$$a_2 = 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1$$
$$a_3 = 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0$$
$$a_4 = 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1$$

$$a_1 = 1, \bar{1}, 1, \bar{1}, 1, 1, \bar{1}, 1, \bar{1}, 1, \bar{1}, \bar{1}, \bar{1}, \bar{1}, 1, \bar{1}, 1, \bar{1}, 1, 1, \bar{1}, \bar{1}, \bar{1}, 1, \bar{1}, \bar{1}, 1, 1, 1, \bar{1}, \bar{1}, 1, 1, \bar{1}, \bar{1}, 1, 1, 1, 1, 1, 1, \bar{1}, \bar{1}, 1, 1, 1, 1, 1, \bar{1}, \bar{1}, \bar{1}, \bar{1}, 1, \bar{1}, 1, \bar{1}, \bar{1}, \bar{1}, \bar{1}, 1, \bar{1}, 1, \bar{1}, \bar{1}, \bar{1}$$
$$a_2 = 1, \bar{1}, 0, 0, 0, 1, 0, 0, \bar{1}, 1, 0, \bar{1}, \bar{1}, 0, 1, 0, 0, 0, 1, 1, \bar{1}, 0, \bar{1}, 1, 0, \bar{1}, 0, 0, 1, 0, \bar{1}, 1, 1, 0, \bar{1}, 1, 0, 0, 1, 1, 1, \bar{1}, \bar{1}, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, \bar{1}, \bar{1}, 0, 0, 1, \bar{1}, 0, 0, \bar{1}, \bar{1}$$
$$a_3 = 0, 0, 1, 0, 1, 0, \bar{1}, 1, 0, 0, \bar{1}, 0, 0, 0, 1, 0, 0, 0, 0, 1, \bar{1}, \bar{1}, \bar{1}, 0, \bar{1}, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, \bar{1}, 0, \bar{1}, 0, 0, 1, \bar{1}, 0, 0, 0, 1, \bar{1}, 1, \bar{1}, 0, 0$$
$$a_4 = 1, \bar{1}, 0, \bar{1}, 1, 1, \bar{1}, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, \bar{1}, 0, 0, 0, 0, \bar{1}, 0, 0, 1, \bar{1}, 0, 1, 0, \bar{1}, \bar{1}, 0, 1, 0, 0, 0, 1, \bar{1}, 0, 0, 0, 1, 1, 1, \bar{1}, \bar{1}, \bar{1}, \bar{1}, 0, \bar{1}, 1, 0, \bar{1}, \bar{1}, 0, 0, 0, 0, 0, \bar{1}, \bar{1}$$

Multi-Scalar Recoding

Step 1: recode a_1 to signed non-zero representation

Step 2: recode a_2, a_3 and a_4 by “sign-aligning” columns

$$
\begin{aligned}
a_1 &= 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1\\
a_2 &= 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1\\
a_3 &= 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0\\
a_4 &= 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1
\end{aligned}
$$

$$
\begin{aligned}
a_1 &= 1, \bar{1}, 1, \bar{1}, 1, 1, \bar{1}, 1, \bar{1}, 1, \bar{1}, \bar{1}, \bar{1}, \bar{1}, 1, \bar{1}, 1, \bar{1}, 1, 1, \bar{1}, \bar{1}, \bar{1}, 1, \bar{1}, \bar{1}, 1, 1, 1, \bar{1}, \bar{1}, 1, 1, \bar{1}, \bar{1}, 1, 1, 1, 1, 1, 1, \bar{1}, \bar{1}, 1, 1, 1, 1, 1, \bar{1}, \bar{1}, \bar{1}, \bar{1}, 1, \bar{1}, 1, \bar{1}, \bar{1}, \bar{1}, \bar{1}, 1, \bar{1}, 1, \bar{1}, \bar{1}, \bar{1}\\
a_2 &= 1, \bar{1}, 0, 0, 0, 1, 0, 0, \bar{1}, 1, 0, \bar{1}, \bar{1}, 0, 1, 0, 0, 0, 1, 1, \bar{1}, 0, \bar{1}, 1, 0, \bar{1}, 0, 0, 1, 0, \bar{1}, 1, 1, 0, \bar{1}, 1, 0, 0, 1, 1, 1, \bar{1}, \bar{1}, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, \bar{1}, \bar{1}, 0, 0, 1, \bar{1}, 0, 0, \bar{1}, \bar{1}\\
a_3 &= 0, 0, 1, 0, 1, 0, \bar{1}, 1, 0, 0, \bar{1}, 0, 0, 0, 1, 0, 0, 0, 0, 1, \bar{1}, \bar{1}, \bar{1}, 0, \bar{1}, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, \bar{1}, 0, \bar{1}, 0, 0, 1, \bar{1}, 0, 0, 0, 1, \bar{1}, 1, \bar{1}, 0, 0\\
a_4 &= 1, \bar{1}, 0, \bar{1}, 1, 1, \bar{1}, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, \bar{1}, 0, 0, 0, 0, \bar{1}, 0, 0, 1, \bar{1}, 0, 1, 0, \bar{1}, \bar{1}, 0, 1, 0, 0, 0, 1, \bar{1}, 0, 0, 0, 1, 1, 1, \bar{1}, \bar{1}, \bar{1}, \bar{1}, 0, \bar{1}, 1, 0, \bar{1}, \bar{1}, 0, 0, 0, 0, 0, \bar{1}, \bar{1}
\end{aligned}
$$

column signs s_i $\left[+ - + - + + - + - + - - - - + - + - + + - - - + - - + + + - - + + - - + + + + + + - - + + + + + - - - - + - + - - - - + - + - - -\right]$

digits d_i $\left[6, 6, 3, 5, 7, 6, 7, 3, 2, 2, 3, 2, 2, 1, 8, 1, 5, 1, 6, 8, 8, 3, 4, 2, 3, 6, 3, 1, 6, 5, 2, 6, 4, 5, 6, 2, 5, 1, 4, 2, 8, 6, 2, 2, 2, 8, 7, 8, 5, 7, 5, 7, 2, 5, 8, 4, 6, 5, 1, 4, 4, 3, 3, 6, 6\right]$

Regular Multi-Scalar Multiplication

column signs s_i $\left[+ - + - + + - + - + - - - - + - + - + + - - - + - - + + + - - + + - - + + + + + + - - + + + + + - - - - + - + - - - - + - + - - - \right]$

digits d_i $[6, 6, 3, 5, 7, 6, 7, 3, 2, 2, 3, 2, 2, 1, 8, 1, 5, 1, 6, 8, 8, 3, 4, 2, 3, 6, 3, 1, 6, 5, 2, 6, 4, 5, 6, 2, 5, 1, 4, 2, 8, 6, 2, 2, 2, 8, 7, 8, 5, 7, 5, 7, 2, 5, 8, 4, 6, 5, 1, 4, 4, 3, 3, 6, 6]$

Execution

➔ *Load* $Q = T[6] = P + \phi(P) + \psi(\phi(P))$

64 times:

➔ $Q = 2Q - T[6] = P + \phi(P) + \psi(\phi(P))$

➔ $Q = 2Q + T[3] = 3P + 2\phi(P) + \psi(P) + 2\psi(\phi(P))$

➔ $Q = 2Q - T[5] = 5P + 4\phi(P) + 2\psi(P) + 3\psi(\phi(P))$

$\vdots$

T[1]	P
T[2]	$P + \phi(P)$
T[3]	$P + \psi(P)$
T[4]	$P + \phi(P) + \psi(P)$
T[5]	$P + \psi(\phi(P))$
T[6]	$P + \phi(P) + \psi(\phi(P))$
T[7]	$P + \psi(P) + \psi(\phi(P))$
T[8]	$P + \phi(P) + \psi(P) + \psi(\phi(P))$

- **Regular execution** (exactly 64 DBLS and 64 ADDs) facilitates protection against timing/SSCA attacks.
- Reduced number of precomputations (only 8 points).

Regular Multi-Scalar Multiplication

$$\begin{array}{ll} \text{column signs } s_i & \left[+ - + - + + - + - + - - - - + - + - + + - - - + - - + + + - - + + - - + + + + + + - - + + + + + - - - - + - + - - - - + - + - - - \right] \\ \text{digits } d_i & \left[\cancel{6}, 6, 3, 5, 7, 6, 7, 3, 2, 2, 3, 2, 2, 1, 8, 1, 5, 1, 6, 8, 8, 3, 4, 2, 3, 6, 3, 1, 6, 5, 2, 6, 4, 5, 6, 2, 5, 1, 4, 2, 8, 6, 2, 2, 2, 8, 7, 8, 5, 7, 5, 7, 2, 5, 8, 4, 6, 5, 1, 4, 4, 3, 3, 6, 6 \right] \end{array}$$

Execution

➔ *Load* $Q = T[6] = P + \phi(P) + \psi(\phi(P))$

64 times
➔ $Q = 2Q - T[6] = P + \phi(P) + \psi(\phi(P))$
➔ $Q = 2Q + T[3] = 3P + 2\phi(P) + \psi(P) + 2\psi(\phi(P))$
➔ $Q = 2Q - T[5] = 5P + 4\phi(P) + 2\psi(P) + 3\psi(\phi(P))$
⋮

T[1]	P
T[2]	$P + \phi(P)$
T[3]	$P + \psi(P)$
T[4]	$P + \phi(P) + \psi(P)$
T[5]	$P + \psi(\phi(P))$
T[6]	$P + \phi(P) + \psi(\phi(P))$
T[7]	$P + \psi(P) + \psi(\phi(P))$
T[8]	$P + \phi(P) + \psi(P) + \psi(\phi(P))$

- **Regular execution** (exactly 64 DBLS and 64 ADDs) facilitates protection against timing/SSCA attacks.
- Reduced number of precomputations (only 8 points).

Regular Multi-Scalar Multiplication

$$\begin{array}{l} \text{column signs } s_i \\ \text{digits } d_i \end{array} \begin{array}{l} [+ - + - + + - + - + - - - - + - + - + + - - - + - - + + + - - + + - - + + + + + + - - + + + + + - - - - + - + - - - - + - + - - -] \\ [\cancel{6}, \cancel{6}, 3, 5, 7, 6, 7, 3, 2, 2, 3, 2, 2, 1, 8, 1, 5, 1, 6, 8, 8, 3, 4, 2, 3, 6, 3, 1, 6, 5, 2, 6, 4, 5, 6, 2, 5, 1, 4, 2, 8, 6, 2, 2, 2, 8, 7, 8, 5, 7, 5, 7, 2, 5, 8, 4, 6, 5, 1, 4, 4, 3, 3, 6, 6] \end{array}$$

Execution

➔ *Load* $Q = T[6] = P + \phi(P) + \psi(\phi(P))$

64 times

➔ $Q = 2Q - T[6] = P + \phi(P) + \psi(\phi(P))$

➔ $Q = 2Q + T[3] = 3P + 2\phi(P) + \psi(P) + 2\psi(\phi(P))$

➔ $Q = 2Q - T[5] = 5P + 4\phi(P) + 2\psi(P) + 3\psi(\phi(P))$

⋮

T[1]	P
T[2]	$P + \phi(P)$
T[3]	$P + \psi(P)$
T[4]	$P + \phi(P) + \psi(P)$
T[5]	$P + \psi(\phi(P))$
T[6]	$P + \phi(P) + \psi(\phi(P))$
T[7]	$P + \psi(P) + \psi(\phi(P))$
T[8]	$P + \phi(P) + \psi(P) + \psi(\phi(P))$

- **Regular execution** (exactly 64 DBLS and 64 ADDs) facilitates protection against timing/SSCA attacks.
- Reduced number of precomputations (only 8 points).

Regular Multi-Scalar Multiplication

column signs s_i $[+ - + - + + - + - + - - - - + - + - + + - - - + - - + + + - - + + - - + + + + + + - - + + + + + - - - - + - + - - - - + - + - - -]$

digits d_i $[\cancel{6}, \cancel{6}, \cancel{3}, 5, 7, 6, 7, 3, 2, 2, 3, 2, 2, 1, 8, 1, 5, 1, 6, 8, 8, 3, 4, 2, 3, 6, 3, 1, 6, 5, 2, 6, 4, 5, 6, 2, 5, 1, 4, 2, 8, 6, 2, 2, 2, 8, 7, 8, 5, 7, 5, 7, 2, 5, 8, 4, 6, 5, 1, 4, 4, 3, 3, 6, 6]$

Execution

➔ *Load* $Q = T[6] = P + \phi(P) + \psi(\phi(P))$

64 times

➔ $Q = 2Q - T[6] = P + \phi(P) + \psi(\phi(P))$

➔ $Q = 2Q + T[3] = 3P + 2\phi(P) + \psi(P) + 2\psi(\phi(P))$

➔ $Q = 2Q - T[5] = 5P + 4\phi(P) + 2\psi(P) + 3\psi(\phi(P))$

⋮

T[1]	P
T[2]	$P + \phi(P)$
T[3]	$P + \psi(P)$
T[4]	$P + \phi(P) + \psi(P)$
T[5]	$P + \psi(\phi(P))$
T[6]	$P + \phi(P) + \psi(\phi(P))$
T[7]	$P + \psi(P) + \psi(\phi(P))$
T[8]	$P + \phi(P) + \psi(P) + \psi(\phi(P))$

- **Regular execution** (exactly 64 DBLS and 64 ADDs) facilitates protection against timing/SSCA attacks.
- Reduced number of precomputations (only 8 points).

Regular Multi-Scalar Multiplication

$$\begin{array}{ll} \text{column signs } s_i & \left[+ - + - + + - + - + - - - - + - + - + + - - - + - - + + + - - + + - - + + + + + + - - + + + + + - - - - + - + - - - - + - + - - - \right] \\ \text{digits } d_i & \left[\cancel{6}, \cancel{6}, \cancel{3}, \cancel{5}, 7, 6, 7, 3, 2, 2, 3, 2, 2, 1, 8, 1, 5, 1, 6, 8, 8, 3, 4, 2, 3, 6, 3, 1, 6, 5, 2, 6, 4, 5, 6, 2, 5, 1, 4, 2, 8, 6, 2, 2, 2, 8, 7, 8, 5, 7, 5, 7, 2, 5, 8, 4, 6, 5, 1, 4, 4, 3, 3, 6, 6 \right] \end{array}$$

Execution

➔ *Load* $Q = T[6] = P + \phi(P) + \psi(\phi(P))$

64 *times*

➔ $Q = 2Q - T[6] = P + \phi(P) + \psi(\phi(P))$

➔ $Q = 2Q + T[3] = 3P + 2\phi(P) + \psi(P) + 2\psi(\phi(P))$

➔ $Q = 2Q - T[5] = 5P + 4\phi(P) + 2\psi(P) + 3\psi(\phi(P))$

$\vdots$

T[1]	P
T[2]	$P + \phi(P)$
T[3]	$P + \psi(P)$
T[4]	$P + \phi(P) + \psi(P)$
T[5]	$P + \psi(\phi(P))$
T[6]	$P + \phi(P) + \psi(\phi(P))$
T[7]	$P + \psi(P) + \psi(\phi(P))$
T[8]	$P + \phi(P) + \psi(P) + \psi(\phi(P))$

- **Regular execution** (exactly 64 DBLS and 64 ADDs) facilitates protection against timing/SSCA attacks.
- Reduced number of precomputations (only 8 points).

Regular Multi-Scalar Multiplication

column signs s_i $\left[+ - + - + + - + - + - - - - + - + - + + - - - + - - + + + - - + + - - + + + + + + - - + + + + + - - - - + - + - - - - + - + - - -\right]$

digits d_i $\left[\cancel{6}, \cancel{6}, \cancel{3}, \cancel{5}, 7, 6, 7, 3, 2, 2, 3, 2, 2, 1, 8, 1, 5, 1, 6, 8, 8, 3, 4, 2, 3, 6, 3, 1, 6, 5, 2, 6, 4, 5, 6, 2, 5, 1, 4, 2, 8, 6, 2, 2, 2, 8, 7, 8, 5, 7, 5, 7, 2, 5, 8, 4, 6, 5, 1, 4, 4, 3, 3, 6, 6\right]$

Execution

➔ *Load* $Q = T[6] = P + \phi(P) + \psi(\phi(P))$

64 times

➔ $Q = 2Q - T[6] = P + \phi(P) + \psi(\phi(P))$

➔ $Q = 2Q + T[3] = 3P + 2\phi(P) + \psi(P) + 2\psi(\phi(P))$

➔ $Q = 2Q - T[5] = 5P + 4\phi(P) + 2\psi(P) + 3\psi(\phi(P))$

⋮

T[1]	P
T[2]	$P + \phi(P)$
T[3]	$P + \psi(P)$
T[4]	$P + \phi(P) + \psi(P)$
T[5]	$P + \psi(\phi(P))$
T[6]	$P + \phi(P) + \psi(\phi(P))$
T[7]	$P + \psi(P) + \psi(\phi(P))$
T[8]	$P + \phi(P) + \psi(P) + \psi(\phi(P))$

- **Regular execution** (exactly 64 DBLS and 64 ADDs) facilitates protection against timing/SSCA attacks.
- Reduced number of precomputations (only 8 points).

Fourℚ-based co-factor ECDH key exchange
[Ladd-L-Barnes, 2016]

- Documented on Internet draft “Curve4Q”, draft-ladd-cfrg-4q-00
  https://tools.ietf.org/html/draft-ladd-cfrg-4q-00

- Current version describes case with compressed public keys (32 bytes)
- Describes two implementations of scalar multiplication:
  - Naïve version without endomorphisms
  - High-speed version exploiting endomorphisms

Four$\mathbb{Q}$-based co-factor ECDH key exchange
(using compression)

- Compressed public keys are 32 bytes long.
- Validation ensures that decompressed public keys are on the curve.
- Co-factor killing consists of fixed sequence of 8 DBLs and 2 ADDs; protects against small subgroup attacks.

FourℚQ-based co-factor ECDH key exchange
(using compression)

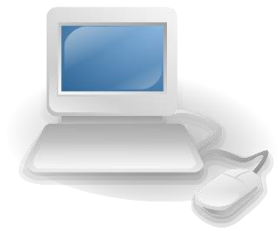
$A = \mathrm{Compress}([a]G)$

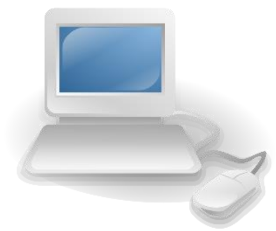
$B = \mathrm{Compress}([b]G)$

- Compressed public keys are 32 bytes long.
- Validation ensures that decompressed public keys are on the curve.
- Co-factor killing consists of fixed sequence of 8 DBLs and 2 ADDs; protects against small subgroup attacks.

FourℚQ-based co-factor ECDH key exchange

(using compression)





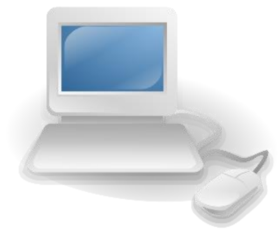
$A = \text{Compress}([a]G)$

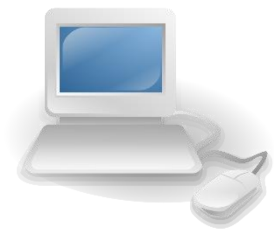
$B = \text{Compress}([b]G)$

$A' = \text{Expand}(A)$

- Compressed public keys are 32 bytes long.
- Validation ensures that decompressed public keys are on the curve.
- Co-factor killing consists of fixed sequence of 8 DBLs and 2 ADDs; protects against small subgroup attacks.

Four$\mathbb{Q}$-based co-factor ECDH key exchange
(using compression)





$A = \text{Compress}([a]G)$

$B = \text{Compress}([b]G)$

$A' = \text{Expand}(A)$
$\text{Validate}(A')$

- Compressed public keys are 32 bytes long.
- Validation ensures that decompressed public keys are on the curve.
- Co-factor killing consists of fixed sequence of 8 DBLs and 2 ADDs; protects against small subgroup attacks.

FourℚQ-based co-factor ECDH key exchange
(using compression)

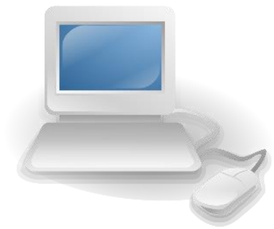
$A = \text{Compress}([a]G)$

$B = \text{Compress}([b]G)$

$A' = \text{Expand}(A)$
$\text{Validate}(A')$
$A'' = [392]A'$

- Compressed public keys are 32 bytes long.
- Validation ensures that decompressed public keys are on the curve.
- Co-factor killing consists of fixed sequence of 8 DBLs and 2 ADDs; protects against small subgroup attacks.

Four$\mathbb{Q}$-based co-factor ECDH key exchange
(using compression)



$A = \text{Compress}([a]G)$

$B = \text{Compress}([b]G)$

$A' = \text{Expand}(A)$
$\text{Validate}(A')$
$A'' = [392]A'$
$S = [b]A'' = [392ab]G$

- Compressed public keys are 32 bytes long.
- Validation ensures that decompressed public keys are on the curve.
- Co-factor killing consists of fixed sequence of 8 DBLs and 2 ADDs; protects against small subgroup attacks.

Four$\mathbb{Q}$-based co-factor ECDH key exchange
(using compression)

$A = \text{Compress}([a]G)$

$B = \text{Compress}([b]G)$

$B' = \text{Expand}(B)$
Validate(B')
$B'' = [392]B'$
$S = [a]B'' = [392ab]G$

$A' = \text{Expand}(A)$
Validate(A')
$A'' = [392]A'$
$S = [b]A'' = [392ab]G$

- Compressed public keys are 32 bytes long.
- Validation ensures that decompressed public keys are on the curve.
- Co-factor killing consists of fixed sequence of 8 DBLs and 2 ADDs; protects against small subgroup attacks.

Four$\mathbb{Q}$-based co-factor ECDH key exchange
(using compression)

$A = \text{Compress}([a]G)$

$B = \text{Compress}([b]G)$

$B' = \text{Expand}(B)$
Validate(B')
$B'' = [392]B'$
$S = [a]B'' = [392ab]G$

$A' = \text{Expand}(A)$
Validate(A')
$A'' = [392]A'$
$S = [b]A'' = [392ab]G$

- Compressed public keys are 32 bytes long.
- Validation ensures that decompressed public keys are on the curve.
- Co-factor killing consists of fixed sequence of 8 DBLs and 2 ADDs; protects against small subgroup attacks.

FourQ-based co-factor ECDH key exchange
(without compression)

$A = [a]G$

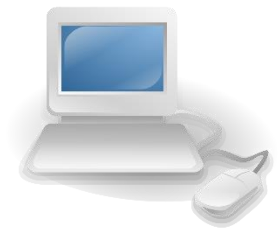
$B = [b]G$

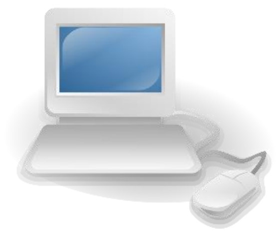
Validate(B)
$B' = [392]B$
$S = [a]B' = [392ab]G$

Validate(A)
$A' = [392]A$
$S = [b]A' = [392ab]G$

- Public keys are 64 bytes long.
- But faster and (slightly) more power efficient.

Four$\mathbb{Q}$-based co-factor ECDH key exchange
(without compression)





$A = [a]G$

$B = [b]G$

Validate(B)
$B' = [392]B$
$S = [a]B' = [392ab]G$

Validate(A)
$A' = [392]A$
$S = [b]A' = [392ab]G$

- Public keys are 64 bytes long.
- But faster and (slightly) more power efficient.

Schnorr$\mathbb{Q}$: a high-speed high-security signature scheme
[Costello-L, 2016]

- Schnorr-type signature scheme closely following EdDSA but based on state-of-the-art curve Four$\mathbb{Q}$
- Optional pre-hashing version (supports single-pass interface for signing)
- Hash-function collision resilience (for version without pre-hashing)
- Deterministic generation
- Small signatures: 64 bytes
- Small public keys: 32 bytes
- *Fastest curve-based signature scheme at the 128-bit level*

E.g., on an Intel Haswell processor:

signing takes 39K cycles (compare to 61K cycles for Ed25519)
verification takes 74K cycles (compare to 185K cycles for Ed25519)

Schnorr$\mathbb{Q}$: a high-speed high-security signature scheme
[Costello-L, 2016]

- Schnorr-type signature scheme closely following EdDSA but based on state-of-the-art curve Four$\mathbb{Q}$
- Optional pre-hashing version (supports single-pass interface for signing)
- Hash-function collision resilience (for version without pre-hashing)
- Deterministic generation
- Small signatures: 64 bytes
- Small public keys: 32 bytes
- *Fastest curve-based signature scheme at the 128-bit level*

E.g., on an Intel Haswell processor:

signing takes 39K cycles (compare to 61K cycles for Ed25519)

verification takes 74K cycles (compare to 185K cycles for Ed25519)

Schnorr$\mathbb{Q}$: a high-speed high-security signature scheme
[Costello-L, 2016]

- Schnorr-type signature scheme closely following EdDSA but based on state-of-the-art curve Four$\mathbb{Q}$
- Optional pre-hashing version (supports single-pass interface for signing)
- Hash-function collision resilience (for version without pre-hashing)
- Deterministic generation
- Small signatures: 64 bytes
- Small public keys: 32 bytes
- *Fastest curve-based signature scheme at the 128-bit level*

E.g., on an Intel Haswell processor:

signing takes 39K cycles (compare to 61K cycles for Ed25519)

verification takes 74K cycles (compare to 185K cycles for Ed25519)

Schnorrℚ: a high-speed high-security signature scheme
[Costello-L, 2016]

- Schnorr-type signature scheme closely following EdDSA but based on state-of-the-art curve Fourℚ
- Optional pre-hashing version (supports single-pass interface for signing)
- Hash-function collision resilience (for version without pre-hashing)
- Deterministic generation
- Small signatures: 64 bytes
- Small public keys: 32 bytes
- *Fastest curve-based signature scheme at the 128-bit level*

E.g., on an Intel Haswell processor:

signing takes 39K cycles (compare to 61K cycles for Ed25519)

verification takes 74K cycles (compare to 185K cycles for Ed25519)

Schnorr$\mathbb{Q}$: a high-speed high-security signature scheme
[Costello-L, 2016]

- Schnorr-type signature scheme closely following EdDSA but based on state-of-the-art curve Four$\mathbb{Q}$
- Optional pre-hashing version (supports single-pass interface for signing)
- Hash-function collision resilience (for version without pre-hashing)
- Deterministic generation
- Small signatures: 64 bytes
- Small public keys: 32 bytes
- *Fastest curve-based signature scheme at the 128-bit level*

E.g., on an Intel Haswell processor:

signing takes 39K cycles (compare to 61K cycles for Ed25519)

verification takes 74K cycles (compare to 185K cycles for Ed25519)

Schnorr$\mathbb{Q}$: a high-speed high-security signature scheme
[Costello-L, 2016]

- Schnorr-type signature scheme closely following EdDSA but based on state-of-the-art curve Four$\mathbb{Q}$
- Optional pre-hashing version (supports single-pass interface for signing)
- Hash-function collision resilience (for version without pre-hashing)
- Deterministic generation
- Small signatures: 64 bytes
- Small public keys: 32 bytes
- *Fastest curve-based signature scheme at the 128-bit level*

E.g., on an Intel Haswell processor:

signing takes 39K cycles (compare to 61K cycles for Ed25519)

verification takes 74K cycles (compare to 185K cycles for Ed25519)

Schnorr$\mathbb{Q}$: a high-speed high-security signature scheme
[Costello-L, 2016]

- Schnorr-type signature scheme closely following EdDSA but based on state-of-the-art curve Four$\mathbb{Q}$
- Optional pre-hashing version (supports single-pass interface for signing)
- Hash-function collision resilience (for version without pre-hashing)
- Deterministic generation
- Small signatures: 64 bytes
- Small public keys: 32 bytes
- *Fastest curve-based signature scheme at the 128-bit level*

E.g., on an Intel Haswell processor:

signing takes 39K cycles (compare to 61K cycles for Ed25519)

verification takes 74K cycles (compare to 185K cycles for Ed25519)

Schnorr$\mathbb{Q}$: a high-speed high-security signature scheme
[Costello-L, 2016]

- Schnorr-type signature scheme closely following EdDSA but based on state-of-the-art curve Four$\mathbb{Q}$
- Optional pre-hashing version (supports single-pass interface for signing)
- Hash-function collision resilience (for version without pre-hashing)
- Deterministic generation
- Small signatures: 64 bytes
- Small public keys: 32 bytes
- *Fastest curve-based signature scheme at the 128-bit level*

E.g., on an Intel Haswell processor:

signing takes 39K cycles (compare to 61K cycles for Ed25519)
verification takes 74K cycles (compare to 185K cycles for Ed25519)

https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/SchnorrQ.pdf

FourQ-based crypto coming to FourQlib

- The upcoming version 3.0 of FourQlib will include:
  - FourQ-based co-factor ECDH
  - SchnorrQ digital signatures
- With the following implementations:
  - A portable C implementation
  - An x64-optimized implementation
  - An optimized implementation for 32-bit platforms
  - An optimized implementation for ARM+NEON platforms
  - An optimized implementation for some 32-bit ARM microcontrollers (e.g., ARM Cortex-M4)

Crypto operations are protected against timing attacks, cache attacks, exception attacks, invalid curve attacks and small subgroup attacks

Four$\mathbb{Q}$-based crypto coming to Four$\mathbb{Q}$lib

- The upcoming version 3.0 of Four$\mathbb{Q}$lib will include:
  - Four$\mathbb{Q}$-based co-factor ECDH
  - Schnorr$\mathbb{Q}$ digital signatures
- With the following implementations:
  - A portable C implementation
  - An x64-optimized implementation
  - An optimized implementation for 32-bit platforms
  - An optimized implementation for ARM+NEON platforms
  - An optimized implementation for some 32-bit ARM microcontrollers (e.g., ARM Cortex-M4)

Crypto operations are protected against timing attacks, cache attacks, exception attacks, invalid curve attacks and small subgroup attacks

Four$\mathbb{Q}$-based crypto coming to Four$\mathbb{Q}$lib

- The upcoming version 3.0 of Four$\mathbb{Q}$lib will include:
  - Four$\mathbb{Q}$-based co-factor ECDH
  - Schnorr$\mathbb{Q}$ digital signatures
- With the following implementations:
  - A portable C implementation
  - An x64-optimized implementation
  - An optimized implementation for 32-bit platforms
  - An optimized implementation for ARM+NEON platforms
  - An optimized implementation for some 32-bit ARM microcontrollers (e.g., ARM Cortex-M4)

Crypto operations are protected against timing attacks, cache attacks, exception attacks, invalid curve attacks and small subgroup attacks

Performance analysis on microcontrollers
[Liu-L-Pereira-Seo, 2016]

- Ported and specialized Four$\mathbb{Q}$lib to various 8-bit and 32-bit microcontrollers:
  - 8-bit AVR ATmega microcontroller
  - 16-bit MSP microcontroller
  - 32-bit ARM Cortex-M4 microcontroller

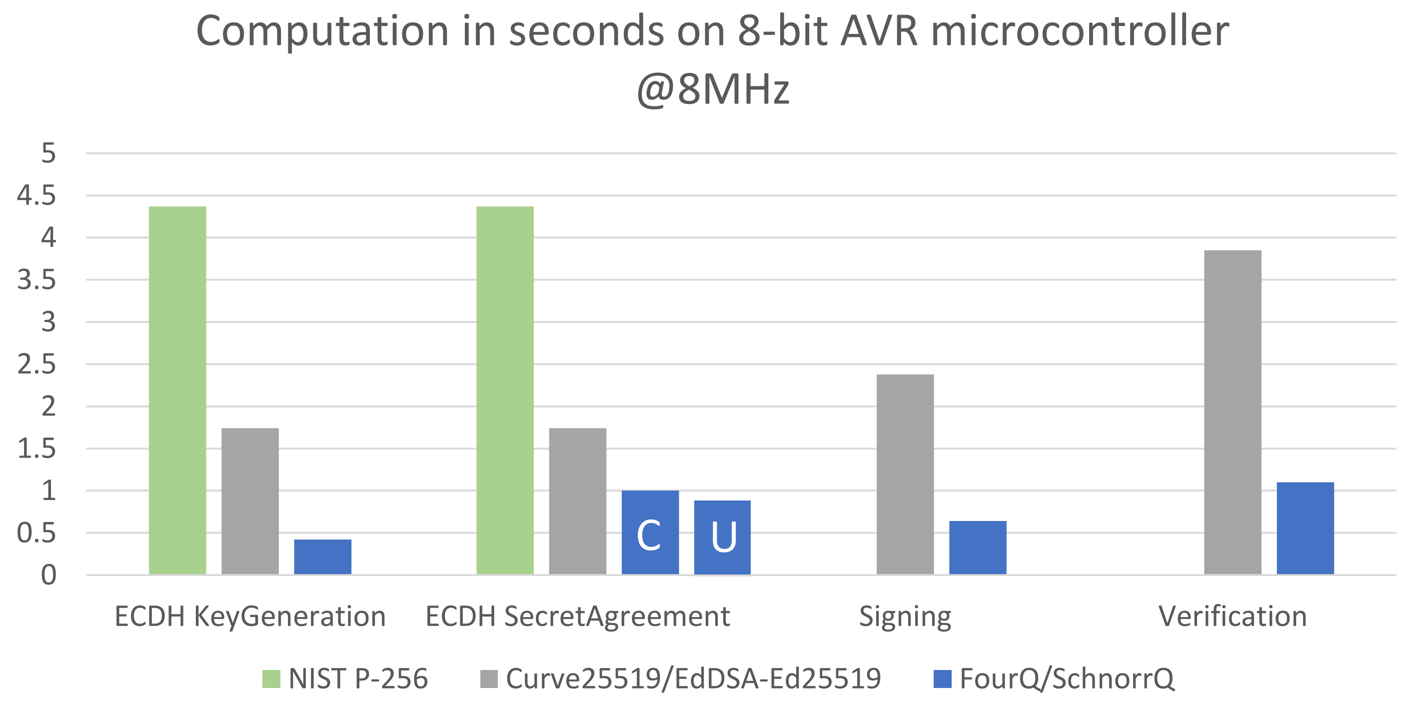
Performance analysis on microcontrollers
[Liu-L-Pereira-Seo, 2016]

- Ported and specialized Four$\mathbb{Q}$lib to various 8-bit and 32-bit microcontrollers:
  - 8-bit AVR ATmega microcontroller
  - 16-bit MSP microcontroller
  - 32-bit ARM Cortex-M4 microcontroller

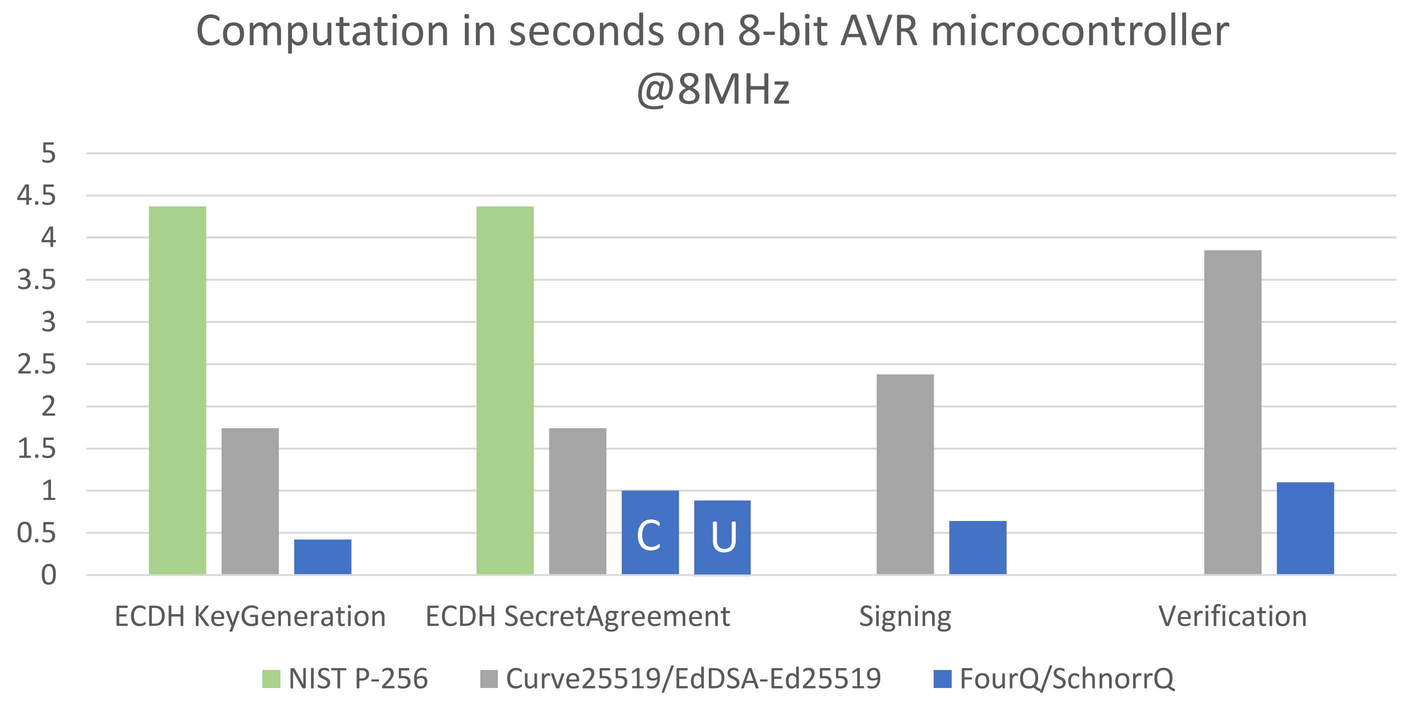
Speed (in thousands of cycles) to compute variable-base scalar multiplication.

Platform	Four$\mathbb{Q}$	Curve25519	Speedup ratio
8-bit AVR ATmega	**6,895**	13,900	**2x**
32-bit ARM Cortex-M4	**531**	1,424	**2.7x**

Performance analysis on AVR microcontroller
[Liu-L-Pereira-Seo, 2016]



Performance analysis on AVR microcontroller
[Liu-L-Pereira-Seo, 2016]



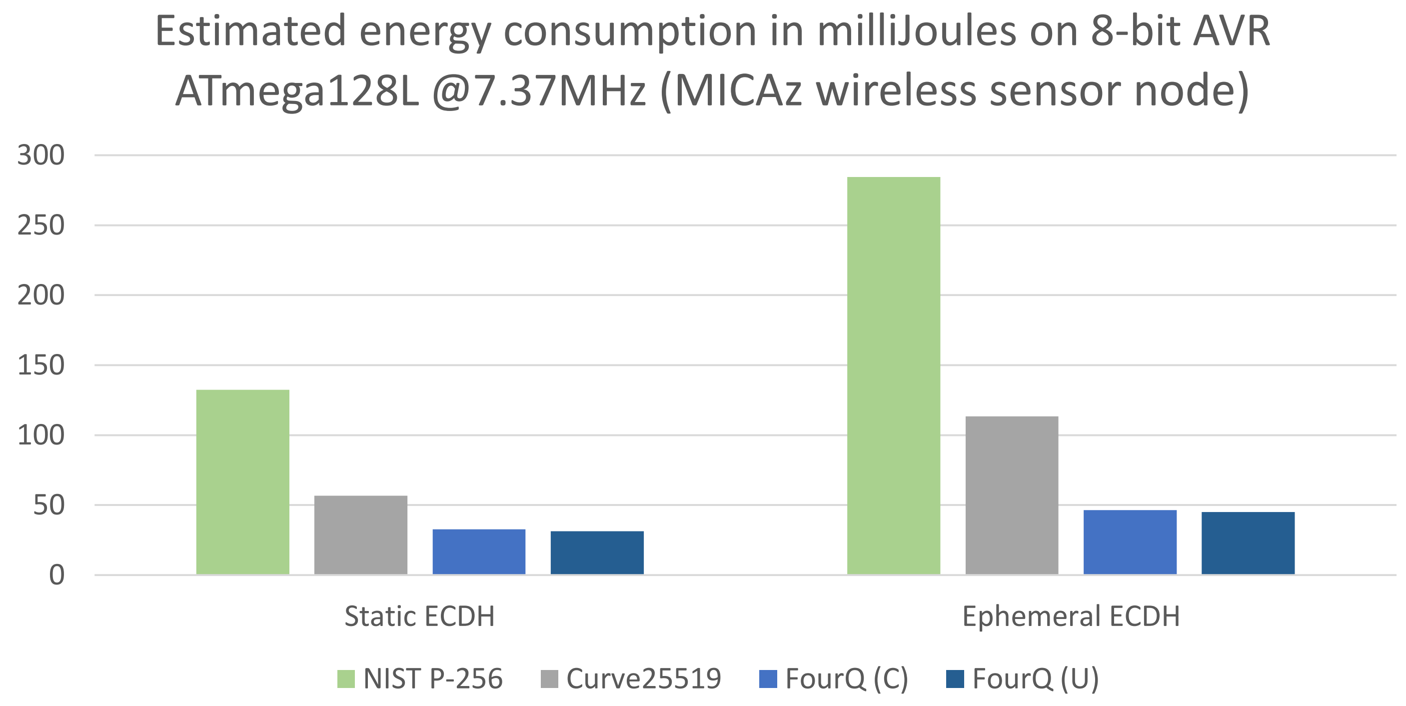
1. Results for ECDH-Four$\mathbb{Q}$ and Schnorr$\mathbb{Q}$ include cost of BLAKE2s for hashing.
2. ECDH-Curve25519 implementation by Düll et al. [DCC 2015].
3. EdDSA-Ed25519-SHA512 implementation by Nascimento-López-Dahab [SPACE 2015].
4. ECDH-NIST-P256 implementation by Wenger et al. [Indocrypt 2013].

(2) and (4):

- Do not exploit fixed-base scalar multiplication.
- Do not include cost of hashing.

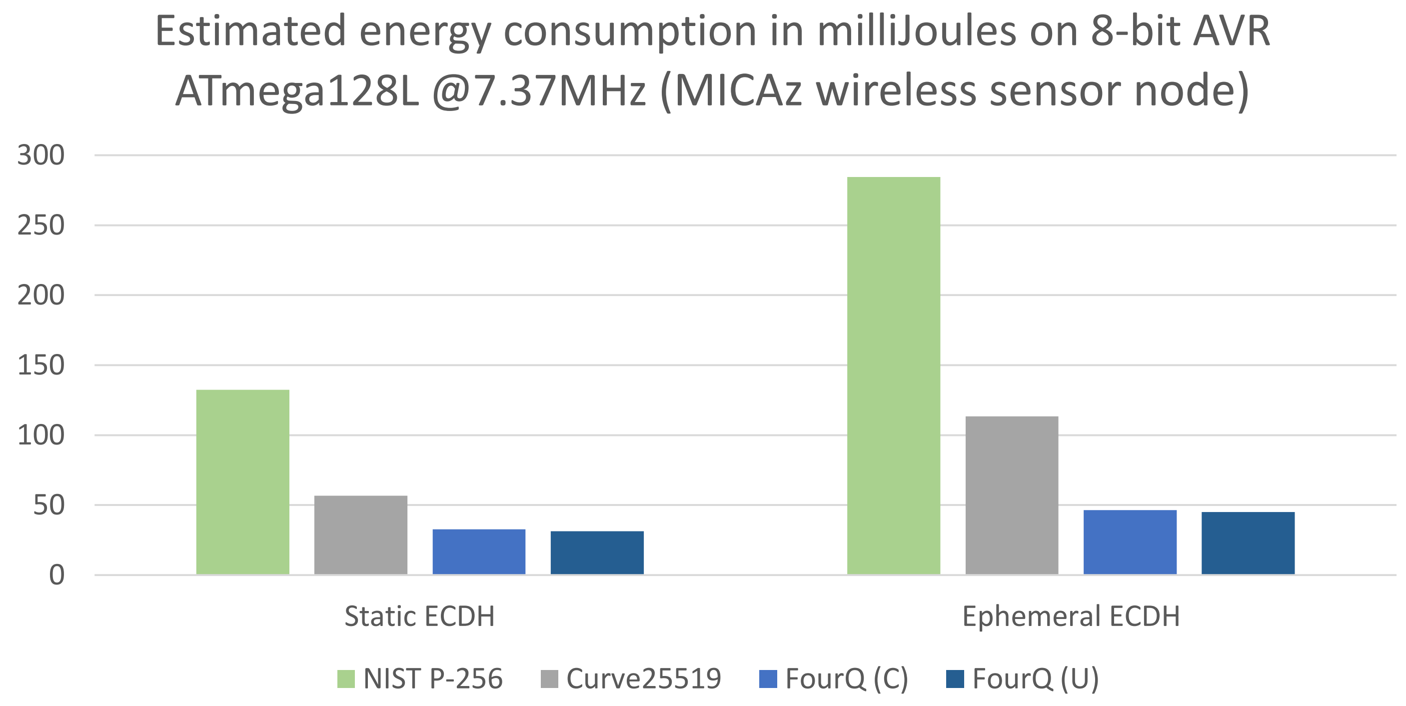
Performance analysis on AVR microcontroller

[Liu-L-Pereira-Seo, 2016]



Performance analysis on AVR microcontroller

[Liu-L-Pereira-Seo, 2016]



1. Results for ECDH-Four$\mathbb{Q}$ and Schnorr$\mathbb{Q}$ include cost of BLAKE2s for hashing.
2. ECDH-Curve25519 implementation by Düll et al. [DCC 2015].
3. EdDSA-Ed25519-SHA512 implementation by Nascimento-López-Dahab [SPACE 2015].
4. ECDH-NIST-P256 implementation by Wenger et al. [Indocrypt 2013].

(2) and (4):

- Do not exploit fixed-base scalar multiplication.
- Do not include cost of hashing.

Performance analysis on AVR microcontroller
[Liu-L-Pereira-Seo, 2016]

- Our implementation prioritizes speed.
- **Trade-off:** much higher speed and reduced energy consumption but higher memory consumption.
- Example: variable-base scalar multiplication requires 35,085 bytes of code versus 17,710 bytes required by Curve25519.

But FourQ is very flexible: one can even use the Montgomery ladder for highly-constrained applications and still be faster and more power-efficient.

Performance analysis on AVR microcontroller
[Liu-L-Pereira-Seo, 2016]

- Our implementation prioritizes speed.
- **Trade-off:** much higher speed and reduced energy consumption but higher memory consumption.
- Example: variable-base scalar multiplication requires 35,085 bytes of code versus 17,710 bytes required by Curve25519.

But Four$\mathbb{Q}$ is very flexible: one can even use the Montgomery ladder for highly-constrained applications and still be faster and more power-efficient.

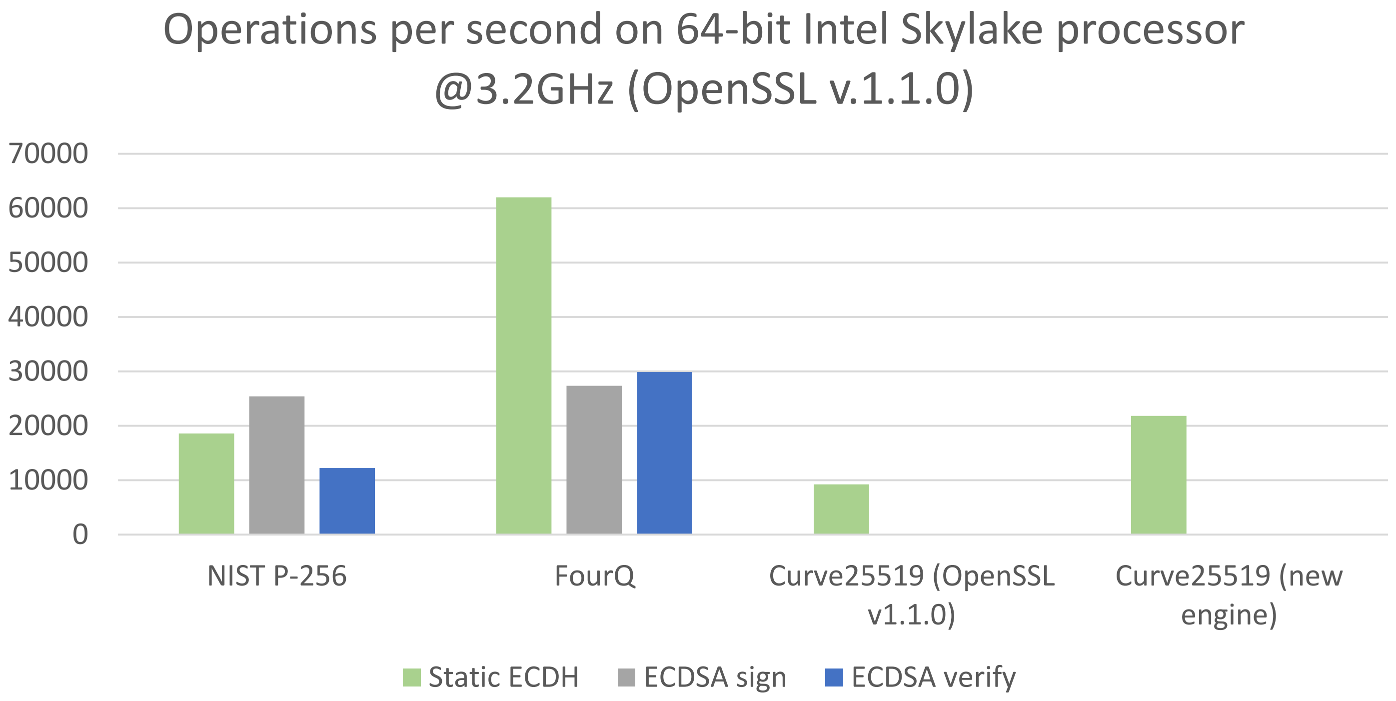
Fourℚ on OpenSSL (in progress)
[Brumley-L-Tuveri]

- Integration to OpenSSL 1.1.0 completed (using Fourℚlib v2.0)
  - Support for any EC protocol available, including ECDH and ECDSA
- Still using original OpenSSL methods for multiprecision operations
- In progress:
  - Add option using an engine to provide Fourℚ externally (this solves most performance degradation issues)
  - Schnorrℚ integration

Fourℚ on OpenSSL (in progress)
[Brumley-L-Tuveri]

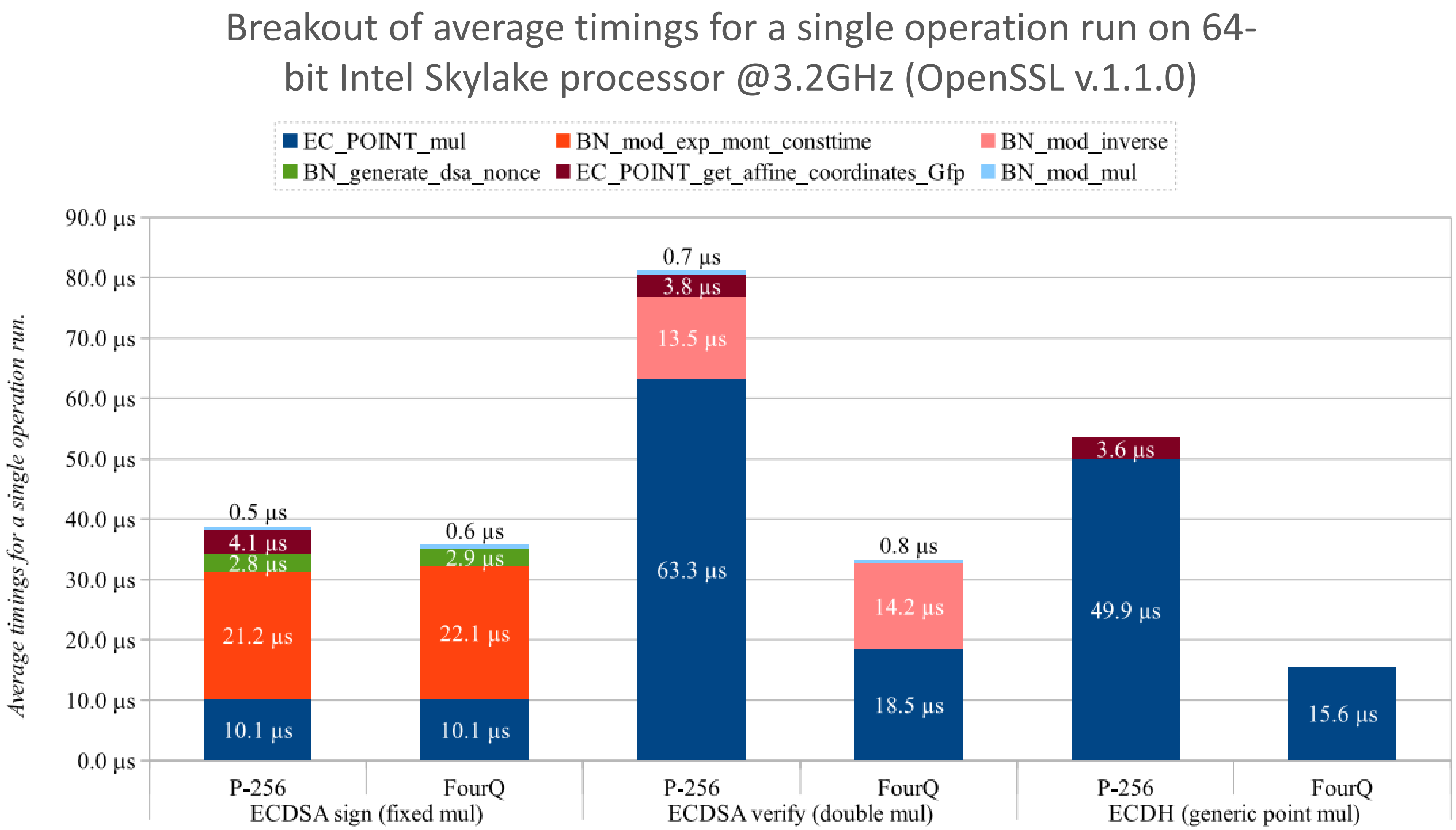
- Integration to OpenSSL 1.1.0 completed (using Fourℚlib v2.0)
  - Support for any EC protocol available, including ECDH and ECDSA
- Still using original OpenSSL methods for multiprecision operations
- In progress:
  - Add option using an engine to provide Fourℚ externally (this solves most performance degradation issues)
  - Schnorrℚ integration

FourQ on OpenSSL (in progress)
[Brumley-L-Tuveri]



- Curve25519's new engine based on Langley's donna_c64 implementation.

FourQ on OpenSSL (in progress)
[Brumley-L-Tuveri]



Additional information

- Four$\mathbb{Q}$ paper: http://eprint.iacr.org/2015/565.pdf
- Four$\mathbb{Q}$lib: https://www.microsoft.com/en-us/research/project/fourqlib/
- RFC draft: https://datatracker.ietf.org/doc/draft-ladd-cfrg-4q/
- Reference implementation in python: https://github.com/bifurcation/fourq
- Schnorr$\mathbb{Q}$: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/SchnorrQ.pdf
- Four$\mathbb{Q}$ on ARM+NEON: http://eprint.iacr.org/2016/645.pdf
- Four$\mathbb{Q}$ on FPGA: http://eprint.iacr.org/2016/569.pdf
- Four$\mathbb{Q}$ on microcontrollers… preprint coming soon!
- Four$\mathbb{Q}$lib version 3.0… release coming soon!
- Four$\mathbb{Q}$ on OpenSSL… release coming soon!

Want to help?

- *Implement Four$\mathbb{Q}$ in Javascript, Rust, Go, etc.*
- *Write code with different speed/simplicity/memory trade-offs on different platforms.*
- *Integrate Four$\mathbb{Q}$ to different cryptographic libraries.*
- *And, ideally, release the code with a friendly open-source license.*

References

[BBJ+08] D.J. Bernstein, P. Birkner, M. Joye, T. Lange and C. Peters. Twisted Edwards curves. AFRICACRYPT 2008.

[BDL+11] D.J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. CHES 2011.

[eBACS] D.J. Bernstein and T. Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems. http://bench.cr.yp.to/results-dh.html

[Edw07] H. Edwards. A normal form for elliptic curves. Bulletin of the AMS, 2007.

[GLS09] S.D. Galbraith, X. Lin, M. Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. EUROCRYPT 2009.

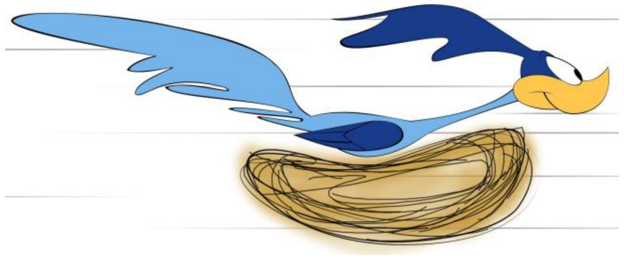
[GLV01] R.P. Gallant, R.J. Lambert, S.A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. CRYPTO 2001.

[GI13] A. Guillevic and S. Ionica. Four-dimensional GLV via the Weil restriction. ASIACRYPT 2013.

[HCW+08] H. Hisil, G. Carter, K.K. Wong and E. Dawson. Twisted Edwards curves revisited. ASIACRYPT 2008.

[Smi13] B. Smith. The Q-curve construction for endomorphism-accelerated elliptic curves. J. Cryptology , 2015.

FourQ-based cryptography for high-performance and low-power applications



Patrick Longa

Microsoft Research

http://research.microsoft.com/en-us/people/plonga/

Microsoft Research