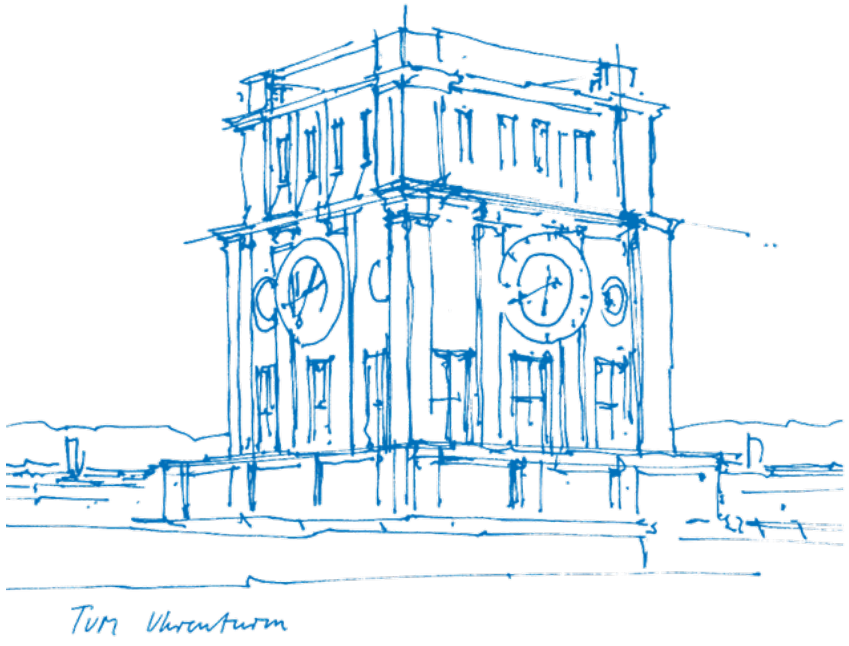
# Towards Side-Channel Protected X25519 on 32-bit ARM Cortex-M4 Embedded Processors

Fabrizio De Santis[1] Georg Sigl[1,2]

[1]Technische Universität München
Faculty of Electrical and Computer Engineering
Institute for Security in Information Technology

[2]Fraunhofer Institute for Applied and
Integrated Security (AISEC), Germany.

SPEED-B, October 19–21, 2016
Utrecht, The Netherlands



# Introduction

There are many ECCs out there ...

# Introduction

There are many ECCs out there ...

# Introduction

There are many ECCs out there ...

- US NIST 186-4 P-256, P-384, P-521, ...

# Introduction

There are many ECCs out there ...

- US NIST 186-4 P-256, P-384, P-521, ...
- US ANSI X9.62 prime256v1, ...

# Introduction

There are many ECCs out there ...

- US NIST 186-4 P-256, P-384, P-521, ...
- US ANSI X9.62 prime256v1, ...
- SECG 2 secp256r1, secp384r1, secp521r1, ...

# Introduction

There are many ECCs out there ...

- US NIST 186-4 P-256, P-384, P-521, ...
- US ANSI X9.62 prime256v1, ...
- SECG 2 secp256r1, secp384r1, secp521r1, ...
- DE Brainpool brainpoolP256r1, brainpoolP384r1, brainpoolP512r1, ...

# Introduction

There are many ECCs out there ...

- US NIST 186-4 P-256, P-384, P-521, ...
- US ANSI X9.62 prime256v1, ...
- SECG 2 secp256r1, secp384r1, secp521r1, ...
- DE Brainpool brainpoolP256r1, brainpoolP384r1, brainpoolP512r1, ...
- Microsoft numsp256t1, numsp384t1, numsp512t1, ...

# Introduction

There are many ECCs out there ...

- US NIST 186-4 P-256, P-384, P-521, ...
- US ANSI X9.62 prime256v1, ...
- SECG 2 secp256r1, secp384r1, secp521r1, ...
- DE Brainpool brainpoolP256r1, brainpoolP384r1, brainpoolP512r1, ...
- Microsoft numsp256t1, numsp384t1, numsp512t1, ...
- ...

# Introduction

There are many ECCs out there ...

- US NIST 186-4 P-256, P-384, P-521, ...
- US ANSI X9.62 prime256v1, ...
- SECG 2 secp256r1, secp384r1, secp521r1, ...
- DE Brainpool brainpoolP256r1, brainpoolP384r1, brainpoolP512r1, ...
- Microsoft numsp256t1, numsp384t1, numsp512t1, ...
- ...
- Aranha et al. M-221, M-383, M-511, ...

# Introduction

There are many ECCs out there ...

- US NIST 186-4 P-256, P-384, P-521, ...
- US ANSI X9.62 prime256v1, ...
- SECG 2 secp256r1, secp384r1, secp521r1, ...
- DE Brainpool brainpoolP256r1, brainpoolP384r1, brainpoolP512r1, ...
- Microsoft numsp256t1, numsp384t1, numsp512t1, ...
- ...
- Aranha et al. M-221, M-383, M-511, ...
- Bernstein et al. Curve25519, Curve41417, ...

# Introduction

There are many ECCs out there ...

- US NIST 186-4 P-256, P-384, P-521, ...
- US ANSI X9.62 prime256v1, ...
- SECG 2 secp256r1, secp384r1, secp521r1, ...
- DE Brainpool brainpoolP256r1, brainpoolP384r1, brainpoolP512r1, ...
- Microsoft numsp256t1, numsp384t1, numsp512t1, ...
- ...
- Aranha et al. M-221, M-383, M-511, ...
- Bernstein et al. Curve25519, Curve41417, ...
- Hamburg Goldilocks448

# Introduction

There are many ECCs out there ...

- US NIST 186-4 P-256, P-384, P-521, ...
- US ANSI X9.62 prime256v1, ...
- SECG 2 secp256r1, secp384r1, secp521r1, ...
- DE Brainpool brainpoolP256r1, brainpoolP384r1, brainpoolP512r1, ...
- Microsoft numsp256t1, numsp384t1, numsp512t1, ...
- ...
- Aranha et al. M-221, M-383, M-511, ...
- Bernstein et al. Curve25519, Curve41417, ...
- Hamburg Goldilocks448
- Costello and Longa FourQ

# Introduction

There are many ECCs out there ...

- US NIST 186-4 P-256, P-384, P-521, ...
- US ANSI X9.62 prime256v1, ...
- SECG 2 secp256r1, secp384r1, secp521r1, ...
- DE Brainpool brainpoolP256r1, brainpoolP384r1, brainpoolP512r1, ...
- Microsoft numsp256t1, numsp384t1, numsp512t1, ...
- ...
- Aranha et al. M-221, M-383, M-511, ...
- Bernstein et al. Curve25519, Curve41417, ...
- Hamburg Goldilocks448
- Costello and Longa FourQ
- ...
- cf. https://safecurves.cr.yp.to/

# Introduction

Most-wanted features:

# Introduction

Most-wanted features:

- **Security**

# Introduction

Most-wanted features:

- **Security**
- **Efficiency** on the widest number of platforms (8-bit, 16-bit, 32-bit, 64-bit)

# Introduction

Most-wanted features:

- **Security**
- **Efficiency** on the widest number of platforms (8-bit, 16-bit, 32-bit, 64-bit)

$\Rightarrow$ Transparent and careful selection of domain parameters, e.g. “rigid generation”.

# Introduction

Most-wanted features:

- **Security**
- **Efficiency** on the widest number of platforms (8-bit, 16-bit, 32-bit, 64-bit)

$\Rightarrow$ Transparent and careful selection of domain parameters, e.g. “rigid generation”.

Focus of this talk:

- High-speed and Compact X25519 for ARM Cortex-M4 processors

# Introduction

Most-wanted features:

- **Security**
- **Efficiency** on the widest number of platforms (8-bit, 16-bit, 32-bit, 64-bit)

$\Rightarrow$ Transparent and careful selection of domain parameters, e.g. “rigid generation”.

Focus of this talk:

- High-speed and Compact X25519 for ARM Cortex-M4 processors

Preliminary results:

- Adding more Side-Channel Protections to X25519

# Introduction

Most-wanted features:

- **Security**
- **Efficiency** on the widest number of platforms (8-bit, 16-bit, 32-bit, 64-bit)

$\Rightarrow$ Transparent and careful selection of domain parameters, e.g. “rigid generation”.

Focus of this talk:

- High-speed and Compact X25519 for ARM Cortex-M4 processors

Preliminary results:

- Adding more Side-Channel Protections to X25519
- Towards Higher Security Levels with X448

# Introduction

Most-wanted features:

- **Security**
- **Efficiency** on the widest number of platforms (8-bit, 16-bit, 32-bit, 64-bit)

$\Rightarrow$ Transparent and careful selection of domain parameters, e.g. “rigid generation”.

Focus of this talk:

- High-speed and Compact X25519 for ARM Cortex-M4 processors

Preliminary results:

- Adding more Side-Channel Protections to X25519
- Towards Higher Security Levels with X448
- ARMing NaCl for Cortex-M4 processors: ChaCha20, Poly1305, ...

# Introduction

Most-wanted features:

- **Security**
- **Efficiency** on the widest number of platforms (8-bit, 16-bit, 32-bit, 64-bit)

$\Rightarrow$ Transparent and careful selection of domain parameters, e.g. “rigid generation”.

Focus of this talk:

- High-speed and Compact X25519 for ARM Cortex-M4 processors

Preliminary results:

- Adding more Side-Channel Protections to X25519
- Towards Higher Security Levels with X448
- ARMing NaCl for Cortex-M4 processors: ChaCha20, Poly1305, ... but also ChaCha20-Poly1305 AEAD

# Curve25519

Montgomery curves:

$$\mathcal{M}/\mathbb{F}_p := \{(x, y) \in \mathbb{F}_p^2 \ : \ By^2 \equiv x^3 + Ax^2 + x \pmod{p}\}$$

Curve25519:

- $p = 2^{255} - 19$, $B = 1$

# Curve25519

Montgomery curves:

$$\mathcal{M}/\mathbb{F}_p := \{(x, y) \in \mathbb{F}_p^2 \ : \ By^2 \equiv x^3 + Ax^2 + x \pmod{p}\}$$

Curve25519:

- $p = 2^{255} - 19$, $B = 1$ , $A = 486662$, $(A + 2)/4 = 121666$.

- Used in many applications, OS, libraries, and protocols like OpenSSH, OpenBSD, Signal, NaCl, BoringSSL*, Tor*, ...
  cf. https://ianix.com/pub/curve25519-deployment.html
- Included in RFC 7748, ...

# Curve25519

Montgomery curves:

$$\mathcal{M}/\mathbb{F}_p := \{(x, y) \in \mathbb{F}_p^2 \ : \ By^2 \equiv x^3 + Ax^2 + x \pmod{p}\}$$

Curve25519:

- $p = 2^{255} - 19$, $B = 1$ , $A = 486662$, $(A + 2)/4 = 121666$.

- Used in many applications, OS, libraries, and protocols like OpenSSH, OpenBSD, Signal, NaCl, BoringSSL*, Tor*, ...
  cf. https://ianix.com/pub/curve25519-deployment.html
- Included in RFC 7748, ...

*Hybrid Post-Quantum Handshake X25519+NewHope:

- Boring SSL under the name CECPQ1 (Google Chrome Canary)
- Tor proposal under the name RebelAlliance

# X25519

X25519 allows to compute a shared secret **K** between two parties $(\alpha, \beta)$ using Curve25519:

$$\alpha = (k_a, \mathbf{P}) \qquad \beta = (k_b, \mathbf{P})$$

# X25519

X25519 allows to compute a shared secret **K** between two parties $(\alpha, \beta)$ using Curve25519:

| $\alpha = (k_a, \mathbf{P})$ | | $\beta = (k_b, \mathbf{P})$ |
|---|---|---|
| $\mathbf{A} = [k_a] \cdot \mathbf{P}$ | $\xrightarrow{\mathbf{A}}$ | |

# X25519

X25519 allows to compute a shared secret **K** between two parties $(\alpha, \beta)$ using Curve25519:

| $\alpha = (k_a, \mathbf{P})$ | | $\beta = (k_b, \mathbf{P})$ |
|---|---|---|
| $\mathbf{A} = [k_a] \cdot \mathbf{P}$ | $\xrightarrow{\mathbf{A}}$ <br> $\xleftarrow{\mathbf{B}}$ | $\mathbf{B} = [k_b] \cdot \mathbf{P}$ |

# X25519

X25519 allows to compute a shared secret **K** between two parties $(\alpha, \beta)$ using Curve25519:

| $\alpha = (k_a, \mathbf{P})$ | | $\beta = (k_b, \mathbf{P})$ |
|---|---|---|
| $\mathbf{A} = [k_a] \cdot \mathbf{P}$ | $\xrightarrow{\mathbf{A}}$ <br> $\xleftarrow{\mathbf{B}}$ | $\mathbf{B} = [k_b] \cdot \mathbf{P}$ |

# X25519

X25519 allows to compute a shared secret **K** between two parties $(\alpha, \beta)$ using Curve25519:

| $\alpha = (k_a, \mathbf{P})$ | | $\beta = (k_b, \mathbf{P})$ |
|---|---|---|
| $\mathbf{A} = [k_a] \cdot \mathbf{P}$ | $\xrightarrow{\mathbf{A}}$ | |
| | $\xleftarrow{\mathbf{B}}$ | $\mathbf{B} = [k_b] \cdot \mathbf{P}$ |
| $\mathbf{K} = [k_a] \cdot \mathbf{B} = [k_a] \cdot [k_b] \cdot \mathbf{P}$ | | $\mathbf{K} = [k_b] \cdot \mathbf{A} = [k_b] \cdot [k_a] \cdot \mathbf{P}$ |

# X25519

X25519 allows to compute a shared secret **K** between two parties $(\alpha, \beta)$ using Curve25519:

| $\alpha = (k_a, \mathbf{P})$ | | $\beta = (k_b, \mathbf{P})$ |
|---|---|---|
| $\mathbf{A} = [k_a] \cdot \mathbf{P}$ | $\xrightarrow{\mathbf{A}}$ | |
| | $\xleftarrow{\mathbf{B}}$ | $\mathbf{B} = [k_b] \cdot \mathbf{P}$ |
| $\mathbf{K} = [k_a] \cdot \mathbf{B} = [k_a] \cdot [k_b] \cdot \mathbf{P}$ | | $\mathbf{K} = [k_b] \cdot \mathbf{A} = [k_b] \cdot [k_a] \cdot \mathbf{P}$ |

Security rests upon ECDLP: X25519 $\approx$ 128-bit security.

# X25519

X25519 allows to compute a shared secret **K** between two parties $(\alpha, \beta)$ using Curve25519:

| $\alpha = (k_a, \mathbf{P})$ | | $\beta = (k_b, \mathbf{P})$ |
|---|---|---|
| $\mathbf{A} = [k_a] \cdot \mathbf{P}$ | $\xrightarrow{\mathbf{A}}$ | |
| | $\xleftarrow{\mathbf{B}}$ | $\mathbf{B} = [k_b] \cdot \mathbf{P}$ |
| $\mathbf{K} = [k_a] \cdot \mathbf{B} = [k_a] \cdot [k_b] \cdot \mathbf{P}$ | | $\mathbf{K} = [k_b] \cdot \mathbf{A} = [k_b] \cdot [k_a] \cdot \mathbf{P}$ |

Security rests upon ECDLP: X25519 $\approx$ 128-bit security.

Scalar multiplication: $\mathbf{Q} = [k] \cdot \mathbf{P} = \mathbf{P} + \mathbf{P} + \cdots + \mathbf{P}$ in the group $(\mathcal{M}/\mathbb{F}_p \cup \mathcal{O}, +)$.

# Montgomery Ladder Algorithm

**Input:** $k = (k_{m-1}, \ldots, k_0)_2 \in \mathbb{N}$, $\mathbf{P} \in \mathcal{M}/\mathbb{F}_p$
**Output:** $\mathbf{Q} = [k] \cdot \mathbf{P} \in \mathcal{M}/\mathbb{F}_p$
$\mathbf{R}_0 \leftarrow \mathcal{O}$; $\mathbf{R}_1 \leftarrow \mathbf{P}$
**for** $i \leftarrow (m-1)$ **downto** 0 **do**
  **if** $k_i == 0$ **then**
    $\mathbf{R}_1 \leftarrow \mathbf{R}_0 + \mathbf{R}_1$ (Point Addition)
    $\mathbf{R}_0 \leftarrow \mathbf{R}_0 + \mathbf{R}_0$ (Point Doubling)
  **else**
    $\mathbf{R}_0 \leftarrow \mathbf{R}_0 + \mathbf{R}_1$ (Point Addition)
    $\mathbf{R}_1 \leftarrow \mathbf{R}_1 + \mathbf{R}_1$ (Point Doubling)
  **end if**
**end for**
**return** $\mathbf{R}_0$

# Montgomery Ladder Algorithm

**Input:** $k = (k_{m-1}, \ldots, k_0)_2 \in \mathbb{N}$, $\mathbf{P} \in \mathcal{M}/\mathbb{F}_p$
**Output:** $\mathbf{Q} = [k] \cdot \mathbf{P} \in \mathcal{M}/\mathbb{F}_p$
$\mathbf{R}_0 \leftarrow \mathcal{O}$; $\mathbf{R}_1 \leftarrow \mathbf{P}$
**for** $i \leftarrow (m-1)$ **downto** 0 **do**
  **if** $k_i == 0$ **then**
    $\mathbf{R}_1 \leftarrow \mathbf{R}_0 + \mathbf{R}_1$ (Point Addition)
    $\mathbf{R}_0 \leftarrow \mathbf{R}_0 + \mathbf{R}_0$ (Point Doubling)
  **else**
    $\mathbf{R}_0 \leftarrow \mathbf{R}_0 + \mathbf{R}_1$ (Point Addition)
    $\mathbf{R}_1 \leftarrow \mathbf{R}_1 + \mathbf{R}_1$ (Point Doubling)
  **end if**
**end for**
**return** $\mathbf{R}_0$

- $x$-coordinate only Montgomery Ladder

# Montgomery Ladder Algorithm

**Input:** $k = (k_{m-1}, \ldots, k_0)_2 \in \mathbb{N}$, $\mathbf{P} \in \mathcal{M}/\mathbb{F}_p$
**Output:** $\mathbf{Q} = [k] \cdot \mathbf{P} \in \mathcal{M}/\mathbb{F}_p$
$\mathbf{R}_0 \leftarrow \mathcal{O}$; $\mathbf{R}_1 \leftarrow \mathbf{P}$
**for** $i \leftarrow (m-1)$ **downto** 0 **do**
  **if** $k_i == 0$ **then**
    $\mathbf{R}_1 \leftarrow \mathbf{R}_0 + \mathbf{R}_1$ (Point Addition)
    $\mathbf{R}_0 \leftarrow \mathbf{R}_0 + \mathbf{R}_0$ (Point Doubling)
  **else**
    $\mathbf{R}_0 \leftarrow \mathbf{R}_0 + \mathbf{R}_1$ (Point Addition)
    $\mathbf{R}_1 \leftarrow \mathbf{R}_1 + \mathbf{R}_1$ (Point Doubling)
  **end if**
**end for**
**return** $\mathbf{R}_0$

- $x$-coordinate only Montgomery Ladder
- Homogeneous Projective Coordinates: $x \mapsto (X, Z)$ such that $x = X/Z$.

# Montgomery Ladder Algorithm

**Input:** $k = (k_{m-1}, \ldots, k_0)_2 \in \mathbb{N}$, $\mathbf{P} \in \mathcal{M}/\mathbb{F}_p$
**Output:** $\mathbf{Q} = [k] \cdot \mathbf{P} \in \mathcal{M}/\mathbb{F}_p$
$\mathbf{R}_0 \leftarrow \mathcal{O}$; $\mathbf{R}_1 \leftarrow \mathbf{P}$
**for** $i \leftarrow (m-1)$ **downto** 0 **do**
  **if** $k_i == 0$ **then**
    $\mathbf{R}_1 \leftarrow \mathbf{R}_0 + \mathbf{R}_1$ (Point Addition)
    $\mathbf{R}_0 \leftarrow \mathbf{R}_0 + \mathbf{R}_0$ (Point Doubling)
  **else**
    $\mathbf{R}_0 \leftarrow \mathbf{R}_0 + \mathbf{R}_1$ (Point Addition)
    $\mathbf{R}_1 \leftarrow \mathbf{R}_1 + \mathbf{R}_1$ (Point Doubling)
  **end if**
**end for**
**return** $\mathbf{R}_0$

- $x$-coordinate only Montgomery Ladder
- Homogeneous Projective Coordinates: $x \mapsto (X, Z)$ such that $x = X/Z$.
- Point addition in $3\mathbf{M} + 2\mathbf{S} + 6\mathbf{A}$, point doubling in $2\mathbf{M} + 2\mathbf{S} + 2\mathbf{A} + 1\mathbf{M}_{121666}$.

# Montgomery Ladder Algorithm

**Input:** $k = (k_{m-1}, \ldots, k_0)_2 \in \mathbb{N}$, $\mathbf{P} \in \mathcal{M}/\mathbb{F}_p$
**Output:** $\mathbf{Q} = [k] \cdot \mathbf{P} \in \mathcal{M}/\mathbb{F}_p$
$\mathbf{R}_0 \leftarrow \mathcal{O}$; $\mathbf{R}_1 \leftarrow \mathbf{P}$
**for** $i \leftarrow (m-1)$ **downto** 0 **do**
  **if** $k_i == 0$ **then**
    $\mathbf{R}_1 \leftarrow \mathbf{R}_0 + \mathbf{R}_1$ (Point Addition)
    $\mathbf{R}_0 \leftarrow \mathbf{R}_0 + \mathbf{R}_0$ (Point Doubling)
  **else**
    $\mathbf{R}_0 \leftarrow \mathbf{R}_0 + \mathbf{R}_1$ (Point Addition)
    $\mathbf{R}_1 \leftarrow \mathbf{R}_1 + \mathbf{R}_1$ (Point Doubling)
  **end if**
**end for**
**return** $\mathbf{R}_0$

- $x$-coordinate only Montgomery Ladder
- Homogeneous Projective Coordinates: $x \mapsto (X, Z)$ such that $x = X/Z$.
- Point addition in $3\mathbf{M} + 2\mathbf{S} + 6\mathbf{A}$, point doubling in $2\mathbf{M} + 2\mathbf{S} + 2\mathbf{A} + 1\mathbf{M}_{121666}$.
- X25519 in $1287\mathbf{M} + 1274\mathbf{S} + 2040\mathbf{A} + 255\mathbf{M}_{121666}$ when inversion in $\mathbb{F}_p$ takes $254\mathbf{S} + 11\mathbf{M}$.

# ARM Cortex M4 Processors

- ARMv7E-M architecture
- 32-bit Thumb®-2 instruction set
- 3-stage pipeline
- $13 + 1$ General-purpose registers
- Optional FPU Unit
- DSP Unit ($32 \times 32$-bit Multiplier :-)

- 32-bit STM32F411RE MCU
- 100 MHz ARM Cortex-M4F
- 512-kB Flash
- 128-kB SRAM
- $I_0 = 100\mu$A/MHz

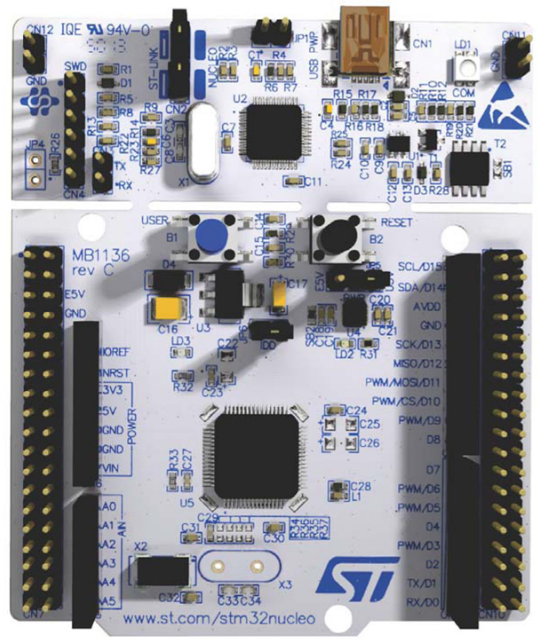



Figure : STMicroelectronics NUCLEO-F411RE

# ARM Cortex M4 Instructions

Usual arithmetic instructions:

- `ADD` $r_2, r_0, r_1$:
  $r_2 = r_0 + r_1$

# ARM Cortex M4 Instructions

Usual arithmetic instructions:

- `ADD` $r_2, r_0, r_1$:
  $r_2 = r_0 + r_1$
- `ADDS` $r_2, r_0, r_1$:
  $r_2 + c_{out} 2^{32} = r_0 + r_1$

# ARM Cortex M4 Instructions

Usual arithmetic instructions:

- `ADD` $r_2, r_0, r_1$:
  $r_2 = r_0 + r_1$
- `ADDS` $r_2, r_0, r_1$:
  $r_2 + c_{out} 2^{32} = r_0 + r_1$
- `ADC` $r_2, r_0, r_1$:
  $r_2 = r_0 + r_1 + c_{in}$

# ARM Cortex M4 Instructions

Usual arithmetic instructions:

- `ADD` $r_2, r_0, r_1$:
  $r_2 = r_0 + r_1$
- `ADDS` $r_2, r_0, r_1$:
  $r_2 + c_{out}2^{32} = r_0 + r_1$
- `ADC` $r_2, r_0, r_1$:
  $r_2 = r_0 + r_1 + c_{in}$
- `ADCS` $r_2, r_0, r_1$:
  $r_2 + c_{out}2^{32} = r_0 + r_1 + c_{in}$

# ARM Cortex M4 Instructions

Usual arithmetic instructions:

- `ADD` $r_2, r_0, r_1$:
  $r_2 = r_0 + r_1$
- `ADDS` $r_2, r_0, r_1$:
  $r_2 + c_{out} 2^{32} = r_0 + r_1$
- `ADC` $r_2, r_0, r_1$:
  $r_2 = r_0 + r_1 + c_{in}$
- `ADCS` $r_2, r_0, r_1$:
  $r_2 + c_{out} 2^{32} = r_0 + r_1 + c_{in}$
- `MUL` $r_2, r_0, r_1$:
  $r_2 = r_0 \cdot r_1$

# ARM Cortex M4 Instructions

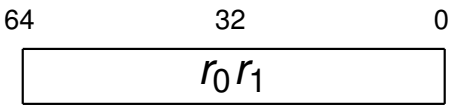
Usual arithmetic instructions:

- `ADD` $r_2, r_0, r_1$:
  $r_2 = r_0 + r_1$
- `ADDS` $r_2, r_0, r_1$:
  $r_2 + c_{out}2^{32} = r_0 + r_1$
- `ADC` $r_2, r_0, r_1$:
  $r_2 = r_0 + r_1 + c_{in}$
- `ADCS` $r_2, r_0, r_1$:
  $r_2 + c_{out}2^{32} = r_0 + r_1 + c_{in}$
- `MUL` $r_2, r_0, r_1$:
  $r_2 = r_0 \cdot r_1$
- ...

# ARM Cortex M4 Instructions

Powerful DSP instructions:

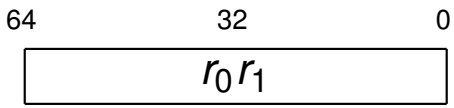
- `UMULL` $r_2, r_3, r_0, r_1$:
  $r_2 + r_3 2^{32} = r_0 r_1$



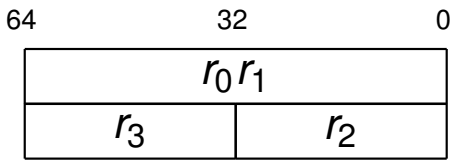
# ARM Cortex M4 Instructions

Powerful DSP instructions:

- `UMULL` $r_2, r_3, r_0, r_1$:
  $r_2 + r_3 2^{32} = r_0 r_1$



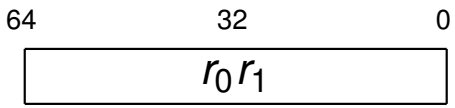

- `UMLAL` $r_2, r_3, r_0, r_1$:
  $r_2 + r_3 2^{32} = r_0 r_1 + (r_2 + r_3 2^{32})$



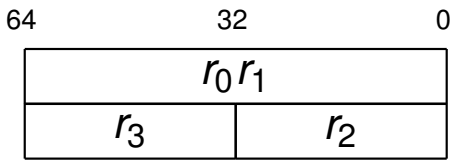
# ARM Cortex M4 Instructions

Powerful DSP instructions:

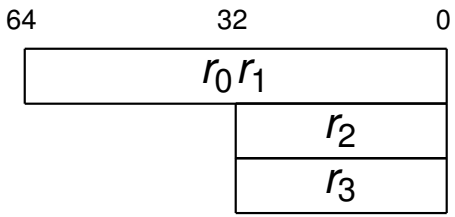
- `UMULL` $r_2, r_3, r_0, r_1$:
  $r_2 + r_3 2^{32} = r_0 r_1$




- `UMLAL` $r_2, r_3, r_0, r_1$:
  $r_2 + r_3 2^{32} = r_0 r_1 + (r_2 + r_3 2^{32})$



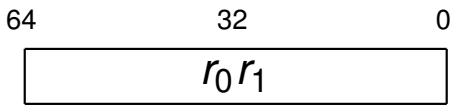

- `UMAAL` $r_2, r_3, r_0, r_1$:
  $r_2 + r_3 2^{32} = r_0 r_1 + (r_2 + r_3)$



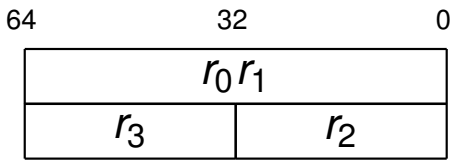
# ARM Cortex M4 Instructions

Powerful DSP instructions:

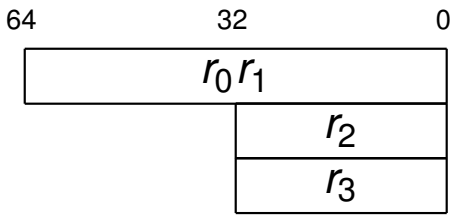
- `UMULL` $r_2, r_3, r_0, r_1$:
  $r_2 + r_3 2^{32} = r_0 r_1$




- `UMLAL` $r_2, r_3, r_0, r_1$:
  $r_2 + r_3 2^{32} = r_0 r_1 + (r_2 + r_3 2^{32})$




- `UMAAL` $r_2, r_3, r_0, r_1$:
  $r_2 + r_3 2^{32} = r_0 r_1 + (r_2 + r_3)$




No carry flags.

# Representation of Integer Numbers and Modular Reduction

- 255-bit integers are represented in radix-$2^{32}$ using 8-limbs:

$$(a_0, \ldots, a_7) \Longleftrightarrow a = \sum_{i=0}^{7} a_i 2^{32i},\ a_i \in \mathbb{Z}_{2^{32}}$$

# Representation of Integer Numbers and Modular Reduction

- 255-bit integers are represented in radix-$2^{32}$ using 8-limbs:

$$(a_0, \ldots, a_7) \Longleftrightarrow a = \sum_{i=0}^{7} a_i 2^{32i},\ a_i \in \mathbb{Z}_{2^{32}}$$

- Fast reduction modulo $2p = 2^{256} - 38$
  - Fit values into 256-bit
  - Aligned to the registers boundaries

# Representation of Integer Numbers and Modular Reduction

- 255-bit integers are represented in radix-$2^{32}$ using 8-limbs:

$$(a_0, \ldots, a_7) \Longleftrightarrow a = \sum_{i=0}^{7} a_i 2^{32i},\ a_i \in \mathbb{Z}_{2^{32}}$$

- Fast reduction modulo $2p = 2^{256} - 38$
  - Fit values into 256-bit
  - Aligned to the registers boundaries

- Full reduction modulo $p = 2^{255} - 19$ at the very end
  - Fit back values to the original field $\mathbb{F}_p$

# Modular Addition/Substraction

1. Straightforward addition with carry (8 `AD?S` instructions):

   **Input:** $a = (a_0, \ldots, a_7)$, $b = (b_0, \ldots, b_7)$.
   **Output:** $c = a + b = (c_0, \ldots, c_7, \gamma_8)$
   $\gamma_0 \leftarrow 0$
   **for** $i \leftarrow 0$ **to** 7 **do**
   $(c_i, \gamma_{i+1}) \leftarrow a_i + b_i + \gamma_i$
   **end for**

# Modular Addition/Substraction

1. Straightforward addition with carry (8 `AD?S` instructions):

   **Input:** $a = (a_0, \ldots, a_7)$, $b = (b_0, \ldots, b_7)$.
   **Output:** $c = a + b = (c_0, \ldots, c_7, \gamma_8)$
   $\gamma_0 \leftarrow 0$
   **for** $i \leftarrow 0$ **to** 7 **do**
   $(c_i, \gamma_{i+1}) \leftarrow a_i + b_i + \gamma_i$
   **end for**

2. Fast reduction by $2p$ (2 `MUL` + 9 `AD?S` instructions):

   **Input:** $c = (c_0, \ldots, c_7, \gamma_8)$
   **Output:** $d \equiv c \bmod 2p$.
   $(d_0, \gamma_1) \leftarrow c_0 + 38\gamma_8$
   **for** $i \leftarrow 1$ **to** 7 **do**
   $(d_i, \gamma_{i+1}) \leftarrow c_i + \gamma_i$
   **end for**
   $d_0 \leftarrow d_0 + 38\gamma_8$

# Modular Addition/Substraction

1. Straightforward addition with carry (8 `AD?S` instructions):

   **Input:** $a = (a_0, \ldots, a_7)$, $b = (b_0, \ldots, b_7)$.
   **Output:** $c = a + b = (c_0, \ldots, c_7, \gamma_8)$
   $\gamma_0 \leftarrow 0$
   **for** $i \leftarrow 0$ **to** 7 **do**
   $(c_i, \gamma_{i+1}) \leftarrow a_i + b_i + \gamma_i$
   **end for**

2. Fast reduction by $2p$ (2 `MUL` + 9 `AD?S` instructions):

   **Input:** $c = (c_0, \ldots, c_7, \gamma_8)$
   **Output:** $d \equiv c \bmod 2p$.
   $(d_0, \gamma_1) \leftarrow c_0 + 38\gamma_8$
   **for** $i \leftarrow 1$ **to** 7 **do**
   $(d_i, \gamma_{i+1}) \leftarrow c_i + \gamma_i$
   **end for**
   $d_0 \leftarrow d_0 + 38\gamma_8$

Total: 106 cycles in 138 bytes.

# $256 \times 256$-bit Multiplication/Squaring

Subtractive Karatsuba:

$$\begin{aligned} ab &= (a_0 + a_1 2^{n/2})(b_0 + b_1 2^{n/2}) \\ &= a_0 b_0 + [(-1)^{(1-t)} |a_0 - a_1||b_0 - b_1| + a_1 b_1 + a_0 b_0] 2^{n/2} + a_1 b_1 2^n \end{aligned}$$

# $256 \times 256$-bit Multiplication/Squaring

Subtractive Karatsuba:

$$\begin{aligned} ab &= (a_0 + a_1 2^{n/2})(b_0 + b_1 2^{n/2}) \\ &= a_0 b_0 + [(-1)^{(1-t)} |a_0 - a_1||b_0 - b_1| + a_1 b_1 + a_0 b_0] 2^{n/2} + a_1 b_1 2^n \end{aligned}$$

Costs:

- 3 multiplications
- 2 additions + 2 subtractions + some shifting
- 2 absolute differences and 1 conditional negation

# 256 × 256-bit Multiplication/Squaring

Subtractive Karatsuba:

$$\begin{aligned} ab &= (a_0 + a_1 2^{n/2})(b_0 + b_1 2^{n/2}) \\ &= a_0 b_0 + [(-1)^{(1-t)}|a_0 - a_1||b_0 - b_1| + a_1 b_1 + a_0 b_0]2^{n/2} + a_1 b_1 2^n \end{aligned}$$

Costs:

- 3 multiplications
- 2 additions + 2 subtractions + some shifting
- 2 absolute differences and 1 conditional negation

First option:

- 3-level subtractive Karatsuba $\Longrightarrow$ 27× (32 × 32)-bit multiplications

# 256 × 256-bit Multiplication/Squaring

Subtractive Karatsuba:

$$\begin{aligned} ab &= (a_0 + a_1 2^{n/2})(b_0 + b_1 2^{n/2}) \\ &= a_0 b_0 + [(-1)^{(1-t)}|a_0 - a_1||b_0 - b_1| + a_1 b_1 + a_0 b_0]2^{n/2} + a_1 b_1 2^n \end{aligned}$$

Costs:

- 3 multiplications
- 2 additions + 2 subtractions + some shifting
- 2 absolute differences and 1 conditional negation

First option:

- 3-level subtractive Karatsuba $\Longrightarrow$ 27× (32 × 32)-bit multiplications

Second option:

- 2-level subtractive Karatsuba $\Longrightarrow$ 9× (64 × 64)-bit multiplications

# $256 \times 256$-bit Multiplication/Squaring

Subtractive Karatsuba:

$$\begin{aligned} ab &= (a_0 + a_1 2^{n/2})(b_0 + b_1 2^{n/2}) \\ &= a_0 b_0 + [(-1)^{(1-t)}|a_0 - a_1||b_0 - b_1| + a_1 b_1 + a_0 b_0]2^{n/2} + a_1 b_1 2^n \end{aligned}$$

Costs:

- 3 multiplications
- 2 additions + 2 subtractions + some shifting
- 2 absolute differences and 1 conditional negation

First option:

- 3-level subtractive Karatsuba $\Longrightarrow$ $27\times$ $(32 \times 32)$-bit multiplications

Second option:

- 2-level subtractive Karatsuba $\Longrightarrow$ $9\times$ $(64 \times 64)$-bit multiplications ✓

Total: 546 cycles and $1,264$ bytes.

# $64 \times 64$-bit Multiplication/Squaring

$$(a_0 + a_1 2^{32})(b_0 + b_1 2^{32}) = a_0 b_0 + (a_0 b_1 + a_1 b_0)2^{32} + a_1 b_1 2^{64}$$

# 64 × 64-bit Multiplication/Squaring

$$(a_0 + a_1 2^{32})(b_0 + b_1 2^{32}) = a_0 b_0 + (a_0 b_1 + a_1 b_0)2^{32} + a_1 b_1 2^{64}$$

Partial Products (4 × `UMULL`):

$$(d_0, d_1) = a_0 b_0$$
$$(d_2, d_3) = a_0 b_1$$
$$(d_4, d_5) = a_1 b_0$$
$$(d_6, d_7) = a_1 b_1$$

# 64 × 64-bit Multiplication/Squaring

$$(a_0 + a_1 2^{32})(b_0 + b_1 2^{32}) = a_0 b_0 + (a_0 b_1 + a_1 b_0)2^{32} + a_1 b_1 2^{64}$$
$$= d_0 + (d_1 + d_2 + d_4)2^{32} + (d_3 + d_5 + d_6)2^{64} + d_7 2^{96}$$

Partial Products (4 × `UMULL`):

$$(d_0, d_1) = a_0 b_0$$
$$(d_2, d_3) = a_0 b_1$$
$$(d_4, d_5) = a_1 b_0$$
$$(d_6, d_7) = a_1 b_1$$

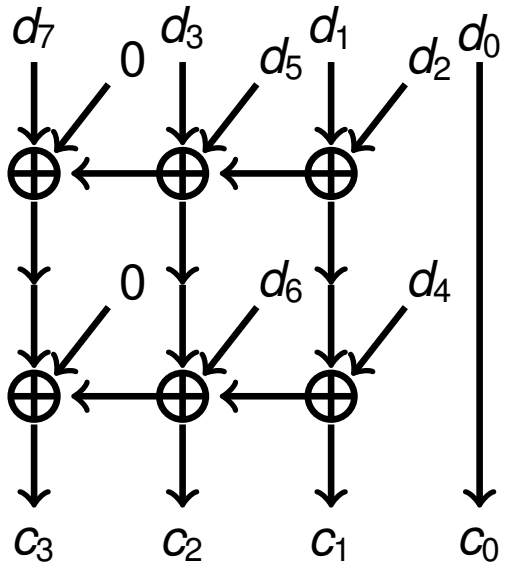
# 64 × 64-bit Multiplication/Squaring

$$(a_0 + a_1 2^{32})(b_0 + b_1 2^{32}) = a_0 b_0 + (a_0 b_1 + a_1 b_0)2^{32} + a_1 b_1 2^{64}$$
$$= d_0 + (d_1 + d_2 + d_4)2^{32} + (d_3 + d_5 + d_6)2^{64} + d_7 2^{96}$$

Partial Products (4 × `UMULL`):

$$(d_0, d_1) = a_0 b_0$$
$$(d_2, d_3) = a_0 b_1$$
$$(d_4, d_5) = a_1 b_0$$
$$(d_6, d_7) = a_1 b_1$$

Adder Tree (2 × `ADDS` + 2 × `ADCS` + 2 × `ADC`):



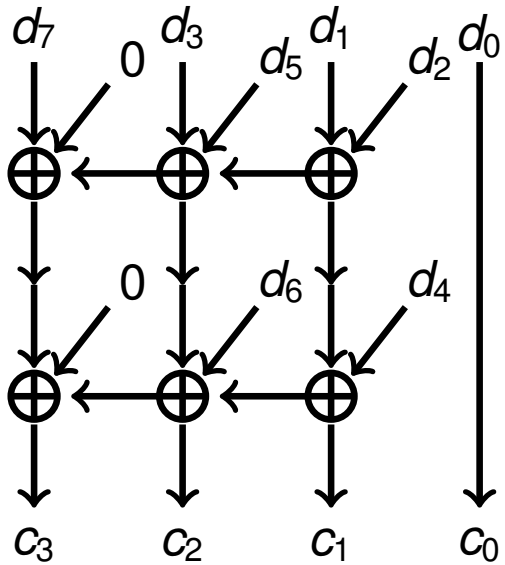
# $64 \times 64$-bit Multiplication/Squaring

$$(a_0 + a_1 2^{32})(b_0 + b_1 2^{32}) = a_0 b_0 + (a_0 b_1 + a_1 b_0)2^{32} + a_1 b_1 2^{64}$$
$$= d_0 + (d_1 + d_2 + d_4)2^{32} + (d_3 + d_5 + d_6)2^{64} + d_7 2^{96}$$

Partial Products ($4 \times$ `UMULL`):

$$(d_0, d_1) = a_0 b_0$$
$$(d_2, d_3) = a_0 b_1$$
$$(d_4, d_5) = a_1 b_0$$
$$(d_6, d_7) = a_1 b_1$$

Adder Tree ($2 \times$ `ADDS` $+ 2 \times$ `ADCS` $+ 2 \times$ `ADC`):




Total: 10 instructions.

# Multiplication by 121666

$121666 \iff \texttt{0x0001db42} \iff 121666 a_i < 2^{49}$

# Multiplication by 121666

$121666 \iff \texttt{0x0001db42} \iff 121666 a_i < 2^{49}$

**Input:** $a = (a_0, \ldots, a_7)$
**Output:** $c = 121666a$.
$c_0 \leftarrow 0$
**for** $i \leftarrow 0$ **to** 7 **do**
    $(c_i, c_{i+1}) \leftarrow 121666 a_i + c_i$
**end for**

# Multiplication by 121666

$121666 \iff \texttt{0x0001db42} \iff 121666a_i < 2^{49}$

**Input:** $a = (a_0, \ldots, a_7)$
**Output:** $c = 121666a$.
$c_0 \leftarrow 0$
**for** $i \leftarrow 0$ **to** 7 **do**
    $(c_i, c_{i+1}) \leftarrow 121666a_i + c_i$
**end for**

`UMAAL` $r_2, r_3, r_0, r_1$:
$r_2 + r_3 2^{32} = r_0 r_1 + (r_2 + r_3)$

# Multiplication by 121666

$121666 \iff \texttt{0x0001db42} \iff 121666a_i < 2^{49}$

**Input:** $a = (a_0, \ldots, a_7)$
**Output:** $c = 121666a$.
$c_0 \leftarrow 0$
**for** $i \leftarrow 0$ **to** 7 **do**
  $(c_i, c_{i+1}) \leftarrow 121666a_i + c_i$
**end for**

`UMAAL` $r_2, r_3, r_0, r_1$:
$r_2 + r_3 2^{32} = r_0 r_1 + (r_2 + r_3)$

First option:
$121666a_i + c_i + 0$

# Multiplication by 121666

$121666 \iff \texttt{0x0001db42} \iff 121666a_i < 2^{49}$

**Input:** $a = (a_0, \ldots, a_7)$
**Output:** $c = 121666a$.
$c_0 \leftarrow 0$
**for** $i \leftarrow 0$ **to** 7 **do**
  $(c_i, c_{i+1}) \leftarrow 121666a_i + c_i$
**end for**

`UMAAL` $r_2, r_3, r_0, r_1$:
$r_2 + r_3 2^{32} = r_0 r_1 + (r_2 + r_3)$

First option:
$121666a_i + c_i + 0$

Second option:
$121665a_i + c_i + a_i$

# Multiplication by 121666

$121666 \iff \texttt{0x0001db42} \iff 121666a_i < 2^{49}$

**Input:** $a = (a_0, \ldots, a_7)$
**Output:** $c = 121666a$.
$c_0 \leftarrow 0$
**for** $i \leftarrow 0$ **to** 7 **do**
  $(c_i, c_{i+1}) \leftarrow 121666a_i + c_i$
**end for**

`UMAAL` $r_2, r_3, r_0, r_1$:
$r_2 + r_3 2^{32} = r_0 r_1 + (r_2 + r_3)$

First option:
$121666a_i + c_i + 0$

Second option:
$121665a_i + c_i + a_i$ ✓

In total: $1\texttt{UMULL} + 7\texttt{UMAAL} + 1\texttt{SUB} = 9$ instructions.

# Implementation Results

$\mathbb{Z}_{2p}$ Arithmetic

| Operation | Speed [Cycles] | Code [Bytes] | Stack [Bytes] |
|---|---|---|---|
| Addition | 106 | 138 | 32 |
| Subtraction | 108 | 148 | 32 |

- GNU Compiler Collection for ARM Embedded Processors version `4.9.3` with `-O2 -mthumb -mcpu=cortex-m4`
- Incl. reduction modulo $2p$ and function call overheads.

# Implementation Results

$\mathbb{Z}_{2p}$ Arithmetic

| Operation | Speed [Cycles] | Code [Bytes] | Stack [Bytes] |
|---|---|---|---|
| Addition | 106 | 138 | 32 |
| Subtraction | 108 | 148 | 32 |
| Multiplication | 546 | 1,264 | 148 |
| Squaring | 362 | 882 | 104 |

- GNU Compiler Collection for ARM Embedded Processors version `4.9.3` with `-O2 -mthumb -mcpu=cortex-m4`
- Incl. reduction modulo $2p$ and function call overheads.

# Implementation Results

$\mathbb{Z}_{2p}$ Arithmetic

| Operation | Speed [Cycles] | Code [Bytes] | Stack [Bytes] |
|---|---|---|---|
| Addition | 106 | 138 | 32 |
| Subtraction | 108 | 148 | 32 |
| Multiplication | 546 | 1,264 | 148 |
| Squaring | 362 | 882 | 104 |
| Multiplication by 121666 | 72 | 116 | 24 |

- GNU Compiler Collection for ARM Embedded Processors version `4.9.3` with `-O2 -mthumb -mcpu=cortex-m4`
- Incl. reduction modulo $2p$ and function call overheads.

# Implementation Results

$\mathbb{Z}_{2p}$ Arithmetic

| Operation | Speed [Cycles] | Code [Bytes] | Stack [Bytes] |
|---|---|---|---|
| Addition | 106 | 138 | 32 |
| Subtraction | 108 | 148 | 32 |
| Multiplication | 546 | 1,264 | 148 |
| Squaring | 362 | 882 | 104 |
| Multiplication by 121666 | 72 | 116 | 24 |
| Inversion (254S+11M) | 96,337 | 484 | 480 |

- GNU Compiler Collection for ARM Embedded Processors version `4.9.3` with `-O2 -mthumb -mcpu=cortex-m4`
- Incl. reduction modulo $2p$ and function call overheads.

# Implementation Results

Variable-base Single-scalar Multiplication

| | Platform | 256 × 256-bit Multiply [Cycles] | 256-bit Square [Cycles] | S/M Ratio | Curve25519 [Cycles] | Curve25519 [Bytes] |
|---|---|---|---|---|---|---|
| 8-bit | AVR ATmega [1] | 6,868 | – | 1 | 22,791,580 | – |
| | AVR ATmega [2] | 7,555 | 5,666 | 0.75 | 20,153,658 | – |
| | AVR ATmega [3] | 4,961 | 3,324 | 0.67 | 13,900,397 | 17,710 |

[1] M. Hutter and P. Schwabe “NaCl on 8-Bit AVR Microcontrollers”, AFRICACRYPT 2013.
[2] E. Nascimento et al. “Efficient and Secure Elliptic Curve Cryptography for 8-bit AVR Microcontrollers”, SPACE 2015.
[3] M. Düll et al. “High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers”, Designs, Codes and Cryptography 2015.

# Implementation Results

Variable-base Single-scalar Multiplication

| | Platform | 256 × 256-bit Multiply [Cycles] | 256-bit Square [Cycles] | S/M Ratio | Curve25519 [Cycles] | Curve25519 [Bytes] |
|---|---|---|---|---|---|---|
| 8-bit | AVR ATmega [1] | 6,868 | – | 1 | 22,791,580 | – |
| | AVR ATmega [2] | 7,555 | 5,666 | 0.75 | 20,153,658 | – |
| | AVR ATmega [3] | 4,961 | 3,324 | 0.67 | 13,900,397 | 17,710 |
| 16-bit | MSP430 [4] | 3,606 | – | 1 | 9,139,739 | 11,778 |
| | MSP430 [3] | 3,193 | 2,426 | 0.76 | 7,933,296 | 13,112 |
| | MSP430 [4] | 2,488 | – | 1 | 6,513,011 | 8,956 |
| | MSP430 [3] | 2,079 | 1,563 | 0.75 | 5,301,792 | 10,088 |


[1] M. Hutter and P. Schwabe “NaCl on 8-Bit AVR Microcontrollers”, AFRICACRYPT 2013.
[2] E. Nascimento et al. “Efficient and Secure Elliptic Curve Cryptography for 8-bit AVR Microcontrollers”, SPACE 2015.
[3] M. Düll et al. “High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers”, Designs, Codes and Cryptography 2015.
[4] G. Hinterwälder et al. “Full-Size High-Security ECC Implementation on MSP430 Microcontrollers”, LATINCRYPT 2014.

# Implementation Results

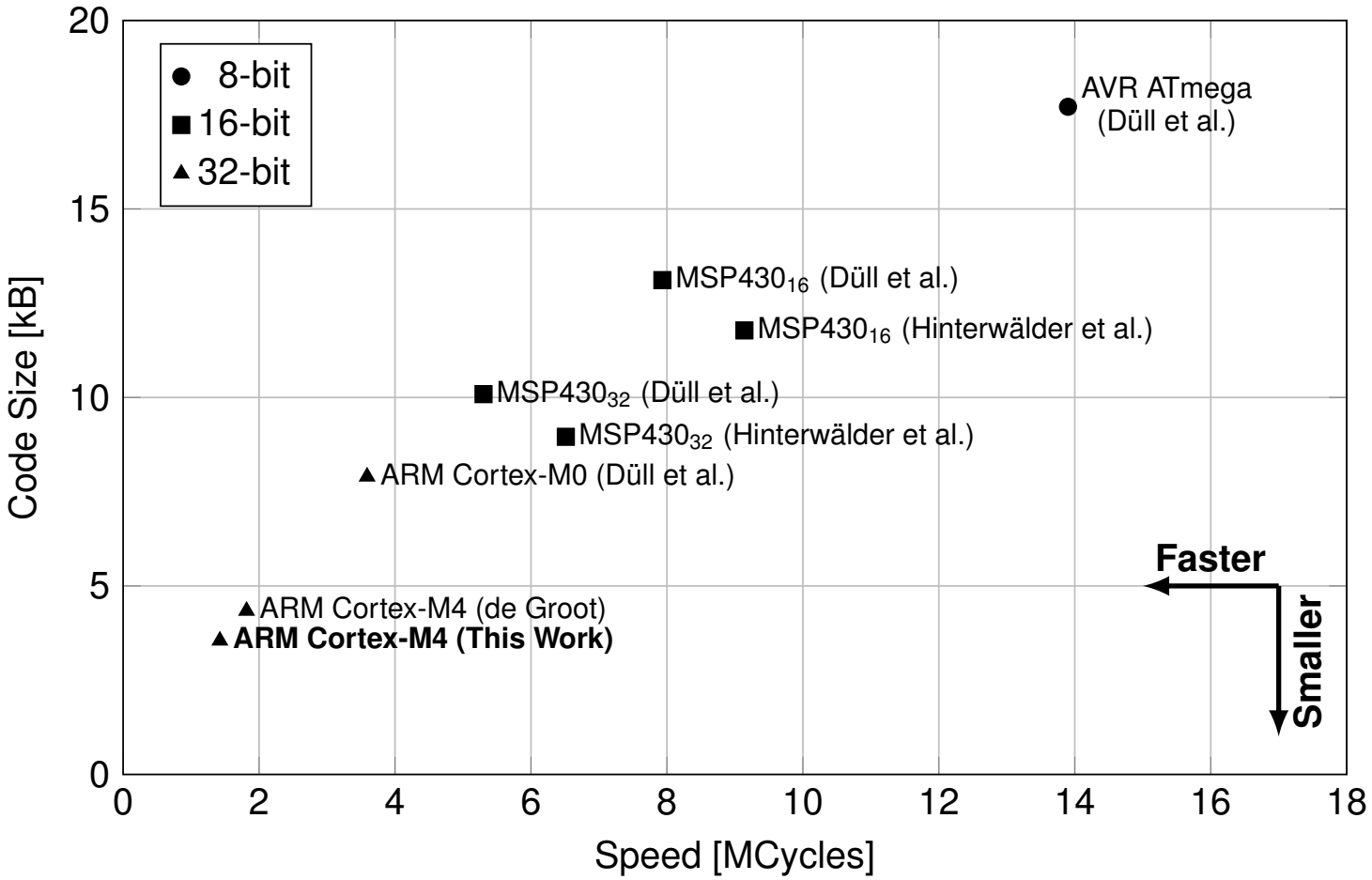
Variable-base Single-scalar Multiplication

| | Platform | 256 × 256-bit Multiply [Cycles] | 256-bit Square [Cycles] | S/M Ratio | Curve25519 [Cycles] | Curve25519 [Bytes] |
|---|---|---|---|---|---|---|
| 8-bit | AVR ATmega [1] | 6,868 | – | 1 | 22,791,580 | – |
| | AVR ATmega [2] | 7,555 | 5,666 | 0.75 | 20,153,658 | – |
| | AVR ATmega [3] | 4,961 | 3,324 | 0.67 | 13,900,397 | 17,710 |
| 16-bit | MSP430 [4] | 3,606 | – | 1 | 9,139,739 | 11,778 |
| | MSP430 [3] | 3,193 | 2,426 | 0.76 | 7,933,296 | 13,112 |
| | MSP430 [4] | 2,488 | – | 1 | 6,513,011 | 8,956 |
| | MSP430 [3] | 2,079 | 1,563 | 0.75 | 5,301,792 | 10,088 |
| 32-bit | ARM Cortex-M0 [3] | 1,294 | 857 | 0.66 | 3,589,850 | 7,900 |
| | ARM Cortex-M4 [5] | 631 | 563 | 0.89 | 1,816,351 | 4,140 |
| | **ARM Cortex-M4 [This Work]** | 546 | – | 1 | **1,658,083** | **2,952** |
| | **ARM Cortex-M4 [This Work]** | 546 | 362 | 0.66 | **1,423,667** | **3,750** |


[1] M. Hutter and P. Schwabe “NaCl on 8-Bit AVR Microcontrollers”, AFRICACRYPT 2013.
[2] E. Nascimento et al. “Efficient and Secure Elliptic Curve Cryptography for 8-bit AVR Microcontrollers”, SPACE 2015.
[3] M. Düll et al. “High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers”, Designs, Codes and Cryptography 2015.
[4] G. Hinterwälder et al. “Full-Size High-Security ECC Implementation on MSP430 Microcontrollers”, LATINCRYPT 2014.
[5] W. de Groot “"A Performance Study of X25519 on Cortex M3 and M4”, Master Thesis 2015.

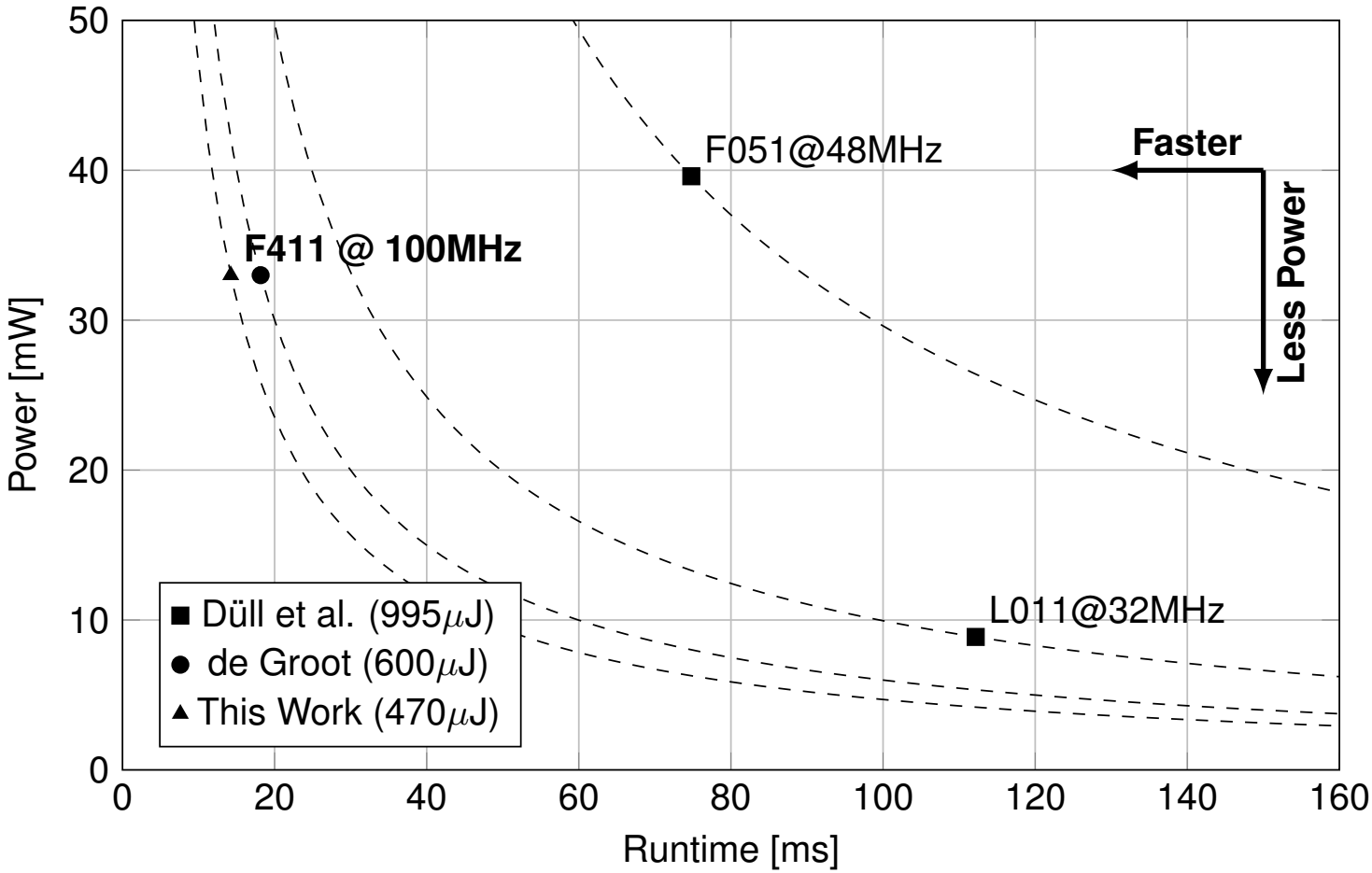
# Implementation Results

Area vs Speed



# Implementation Results

Power/Energy vs Runtime



# Adding more Side-Channel Protections to X25519

## Randomized Projective Coordinates

$x \mapsto (\lambda X, \lambda Z)$ for $\lambda \leftarrow^{\$} \mathbb{F}^*_{2^{255}-19}$

# Adding more Side-Channel Protections to X25519

Randomized Projective Coordinates

$x \mapsto (\lambda X, \lambda Z)$ for $\lambda \leftarrow^{\$} \mathbb{F}^*_{2^{255}-19}$

Costs:

- 1543**M** instead of 1287**M**, i.e. $1,423,667 \rightarrow 1,563,582$ cycles
- Incl. cycles for setting up and generating 64-bytes randomness with ChaCha20

# Adding more Side-Channel Protections to X25519

Randomized Projective Coordinates

$x \mapsto (\lambda X, \lambda Z)$ for $\lambda \leftarrow^{\$} \mathbb{F}^*_{2^{255}-19}$

Costs:

- 1543**M** instead of 1287**M**, i.e. $1,423,667 \rightarrow 1,563,582$ cycles
- Incl. cycles for setting up and generating 64-bytes randomness with ChaCha20

Is this all? cf. https://eprint.iacr.org/2016/923.pdf

# Towards Higher Security Levels

X448

Curve448 (RFC7748):

- $p = 2^{448} - 2^{224} - 1$, $B = 1$, $A = 156326$.

Preliminary results:

# Towards Higher Security Levels

X448

Curve448 (RFC7748):

- $p = 2^{448} - 2^{224} - 1$, $B = 1$, $A = 156326$.

Preliminary results:

- 1-level additive Karatsuba

# Towards Higher Security Levels

X448

Curve448 (RFC7748):

- $p = 2^{448} - 2^{224} - 1$, $B = 1$, $A = 156326$.

Preliminary results:

- 1-level additive Karatsuba
- Reduced-radix $2^{28}$ with fast and lazy reduction

# Towards Higher Security Levels

X448

Curve448 (RFC7748):

- $p = 2^{448} - 2^{224} - 1$, $B = 1$, $A = 156326$.

Preliminary results:

- 1-level additive Karatsuba
- Reduced-radix $2^{28}$ with fast and lazy reduction
- $448 \times 448$-bit Squarer/Multiplication incl. reduction modulo $2^{448} - 2^{224} - 1$:

  $1,087/1,532$ cycles $\Rightarrow$ 1**S**=0.71**M**.

# Towards Higher Security Levels

X448

Curve448 (RFC7748):

- $p = 2^{448} - 2^{224} - 1$, $B = 1$, $A = 156326$.

Preliminary results:

- 1-level additive Karatsuba
- Reduced-radix $2^{28}$ with fast and lazy reduction
- $448 \times 448$-bit Squarer/Multiplication incl. reduction modulo $2^{448} - 2^{224} - 1$:

  $1,087/1,532$ cycles $\Rightarrow$ 1**S**=0.71**M**.

- X448 @ $6,939,815$ cycles $\approx$ 69ms@100MHz

# ARMing NaCl on Cortex-M4

ChaCha20 and Poly1305

NaCl:

- X25519 @ $1,423,667$ cycles in $3,750$ bytes ✓

# ARMing NaCl on Cortex-M4

ChaCha20 and Poly1305

NaCl:

- X25519 @ $1,423,667$ cycles in $3,750$ bytes ✓
- Poly1305 @ 3.6 cycles/byte in 648 bytes ✓

# ARMing NaCl on Cortex-M4

ChaCha20 and Poly1305

NaCl:

- X25519 @ $1,423,667$ cycles in $3,750$ bytes ✓
- Poly1305 @ 3.6 cycles/byte in 648 bytes ✓
- ChaCha20 @ 22 cycles/byte in 696 bytes ✓

# ARMing NaCl on Cortex-M4

ChaCha20 and Poly1305

NaCl:

- X25519 @ $1,423,667$ cycles in $3,750$ bytes ✓
- Poly1305 @ 3.6 cycles/byte in 648 bytes ✓
- ChaCha20 @ 22 cycles/byte in 696 bytes ✓
- Ed25519 @ ... in progress

# ARMing NaCl on Cortex-M4

ChaCha20 and Poly1305

NaCl:

- X25519 @ $1,423,667$ cycles in $3,750$ bytes ✓
- Poly1305 @ 3.6 cycles/byte in 648 bytes ✓
- ChaCha20 @ 22 cycles/byte in 696 bytes ✓
- Ed25519 @ ... in progress

IETF/TLS cipher suite as by RFC7905 and RFC7539:

- ChaCha20-Poly1305 AEAD @ 33.6 cycles/byte in $1,668$ bytes ✓

# Conclusion

High-speed and compact X25519 on ARM Cortex M4 processors

- High-speed full-radix field arithmetic
- Exploit powerful DSP multiplication instructions
- Promising results for high-speed IoT applications

Next steps:

1. Ultimate the porting of NaCl on ARM Cortex M4 processors
2. Validate Side-Channel Protections against actual measurements
3. Evaluate various efficiency-security trade-offs, e.g. X448/Ed448