Towards Side-Channel Protected X25519 on ARM Cortex-M4 Processors

Extended Abstract

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

[1] Technische Universität München (TUM), Germany.
{desantis,sigl}@tum.de
[2] Fraunhofer Institute for Applied and Integrated Security (AISEC), Germany.
georg.sigl@aisec.fraunhofer.de

Abstract. Curve25519 is a prime field Montgomery curve for use with the Elliptic Curve Diffie-Hellman (ECDH) key exchange protocol. In the past few years, Curve25519 has received increasing attention, due to its elegant design geared towards security, high-performance, and transparency. In this work, we present an *high-speed* and side-channel protected Curve25519 variable-base scalar multiplication for ARM Cortex-M4 processors, running in constant-time and using randomized projective coordinates to hinder side-channel attacks.

Keywords: Curve25519, X25519, ECDH, ARM, Cortex, IoT Security

1 X25519

Curve25519 is a 255-bit Montgomery curve defined by the equation $y^2 = x^3 + 486662x^2 + x$ over the prime field $\mathbb{F}_{2^{255}-19}$ [Ber06].

The set of points $\{(x, y) \in \mathbb{F}^2_{2^{255}-19} : y^2 \equiv x^3 + 486662x^2 + x \pmod{2^{255} - 19}\}$ together with the point at infinity $\mathcal{O}$ (neutral element) form an additive abelian group under the point addition operation. The repeated addition of a point $\mathbf{P}$ to itself for k times is called *scalar multiplication* and it is shortly denoted as $\mathbf{Q} = [k] \cdot \mathbf{P}$.

The ECDH-Curve25519 protocol (also known as X25519 [Ber]) allows to compute a shared secret between two parties using two scalar multiplications: (1) each party computes a scalar multiplication on Curve25519 between its 32-byte private key k and a 32-byte public point $\mathbf{P}$; (2) each party transmits the 32-byte output $\mathbf{Q} = [k] \cdot \mathbf{P}$ to the opposite party and computes yet a scalar multiplication on Curve25519 between its private key k and the public output received from the opposite party.

A Curve25519 scalar multiplication can be efficiently computed using the (x-coordinate only) Montgomery ladder and homogeneous projective coordinates [Mon87], i.e. point coordinates are represented by (X, Z) such that $x = X/Z$ with $Z \neq 0$.

Algorithm 1 Curve25519 Scalar Multiplication with Randomized Homogeneous Projective Coordinates.

Input: $k = (k_{255}, \ldots, k_0)_2 \in 2^{254} + 8\{0, \ldots, 2^{251} - 1\}$ and x_P s.t. $\mathbf{P} = (x_P, y_P)$
Output: x_Q s.t. $\mathbf{Q} = [k] \cdot \mathbf{P} = (x_Q, y_Q)$

1: $\lambda \leftarrow^{\$} \mathbb{F}^*_{2^{255}-19}; X_\lambda \leftarrow \lambda x_P; X_1 \leftarrow X_\lambda; Z_1 \leftarrow \lambda$ { **1M** }
2: $X_2 \leftarrow^{\$} \mathbb{F}^*_{2^{255}-19}; Z_2 \leftarrow 0$
3: **for** $i = 254$ **downto** 0 **do**
4: $(X_1, X_2) \leftarrow \mathsf{cswap}(X_1, X_2, (k_i \oplus k_{i+1}))$
5: $(Z_1, Z_2) \leftarrow \mathsf{cswap}(Z_1, Z_2, (k_i \oplus k_{i+1}))$
6: $T_0 \leftarrow X_1 - Z_1$ { **1B** }
7: $T_1 \leftarrow X_2 - Z_2$ { **1B** }
8: $X_2 \leftarrow X_2 + Z_2$ { **1A** }
9: $Z_2 \leftarrow X_1 + Z_1$ { **1A** }
10: $Z_1 \leftarrow T_0 X_2$ { **1M** }
11: $Z_2 \leftarrow Z_2 T_1$ { **1M** }
12: $T_0 \leftarrow T_1^2$ { **1S** }
13: $T_1 \leftarrow X_2^2$ { **1S** }
14: $X_1 \leftarrow Z_1 + Z_2$ { **1A** }
15: $Z_2 \leftarrow Z_1 - Z_2$ { **1B** }
16: $X_2 \leftarrow T_1 T_0$ { **1M** }
17: $T_1 \leftarrow T_1 - T_0$ { **1B** }
18: $Z_2 \leftarrow Z_2^2$ { **1S** }
19: $Z_1 \leftarrow 121666 T_1$ { $\mathbf{1M}_{121666}$ }
20: $X_1 \leftarrow X_1^2$ { **1S** }
21: $X_1 \leftarrow \lambda X_1$ { **1M** }
22: $T_0 \leftarrow T_0 + Z_1$ { **1A** }
23: $Z_1 \leftarrow X_\lambda Z_2$ { **1M** }
24: $Z_2 \leftarrow T_1 T_0$ { **1M** }
25: **end for**
26: $(X_1, X_2) \leftarrow \mathsf{cswap}(X_1, X_2, k_0)$
27: $(Z_1, Z_2) \leftarrow \mathsf{cswap}(Z_1, Z_2, k_0)$
28: $Z_2 \leftarrow Z_2^{-1}$ { 254**S**+11**M** }
29: $x_Q \leftarrow X_2 Z_2$ { **1M** }
30: **return** x_Q

In order to protect X25519 implementations against timing attacks, prime field arithmetic must be implemented in constant-time[3]. In order to offer basic protections against differential power attacks, the initial point coordinates can be randomized using a fresh random value $\lambda \leftarrow^{\$} \mathbb{F}^*_{2^{255}-19}$ for each execution [Cor99], i.e. $(\lambda X, \lambda Z)$. The Montgomery Ladder for Curve25519 using randomized homogeneous projective coordinates is provided in Algorithm 1. It requires 1020**A**+1020**B**+1274**S**+1543**M** field operations to perform a Curve25519

[3] The $\mathsf{cswap}(\cdot, \cdot, \cdot)$ function must also be implemented in constant-time.

scalar multiplication, where the letters $\mathbf{A}, \mathbf{B}, \mathbf{S}, \mathbf{M}$ are used to denote field additions, subtractions, squaring and multiplications, respectively.

2 ARM Cortex-M4 Processors

ARM Cortex-M4 processors are 32-bit RISC processors based on the ARMv7-M architecture and targeting small scale applications, such as microcontrollers.

They are equipped with 13 general-purpose registers, plus the link register (`lr`), the stack pointer (`sp`), and the program counter (`pc`). Note that the link register (`lr`) can be used as a general-purpose register, after that its content has been saved.

Load and store instructions take $n + 1$ clock cycles in general, where n is the number of registers involved in the specific memory instruction, i.e. multiple contiguous data can be read from/write to memory. Note that memory accesses can be sped up by scheduling independent load/store instructions which can take advantage of the (three-stages) pipeline.

The ARMv7-M architecture supports the full set of 32-bit Thumb®-2 instructions, including very powerful *single-cycle* "multiply" and "multiply-and-accumulate" instructions [ARMb]. In particular:

- `UMULL rLO, rHI,` a_i`,` b_j multiplies two unsigned integer words a_i, b_j and stores the 64-bit result into the registers `rLO` and `rHI`.
- `UMLAL rLO, rHI,` a_i`,` b_j multiplies two unsigned integer words a_i, b_j and adds the result to the 64-bit value stored into the registers `rLO` and `rHI`.
- `UMAAL rLO, rHI,` a_i`,` b_j multiplies two unsigned integer words a_i, b_j, adds the 32-bit `rLO` value to the result of the multiplication, adds the 32-bit `rHI` value to the result of the addition, and finally stores the 64-bit result back into the registers `rLO` and `rHI`.

Other relevant arithmetic instructions are `ADD`, `SUB`, `ADC`, and `SBC`, which are used to add, subtract, add with carry and subtract with carry, respectively. If these instructions have to update the condition flags in the status register, then they can be used with the "`S`" suffix, i.e. `ADDS`, `SUBS`, `ADCS`, and `SBCS`, respectively.

3 Field Arithmetic

255-bit integers are represented in little-endian format using radix-2^{32} (full-radix), i.e. every integer a is stored as a $d = \lceil 255/32 \rceil = 8$-dimensional array of 32-bit words $(a_0, \ldots, a_{d-1}) = (a \bmod 2^{32}, \ldots, a/2^{32(d-1)} \bmod 2^{32})$ and represented as $a = \sum_{i=0}^{d-1} a_i 2^{32i}$, $a_i \in \mathbb{Z}_{2^{32}}$.

Modular reduction is performed in two steps, as suggested in [CP06, HMH+15]: (1) a fast reduction modulo $2^{256} - 38$ is performed during the ladder steps to align the words to the boundaries of registers and fit integers into 32-byte. (2) A final reduction modulo $2^{255} - 19$ takes place at the end of the scalar multiplication to reduce the final result back to prime field $\mathbb{F}_{2^{255}-19}$.

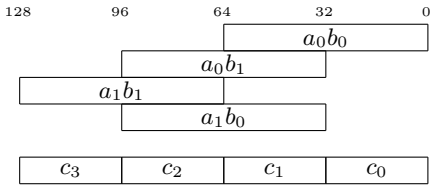

Fig. 1. 8×8-byte Multiplication.

Listing 1 8×8-byte Multiplication.

Input: $a = (a_0, a_1)$ and $b = (b_0, b_1)$.
Output: $c = (c_0, c_1, c_2, c_3)$ s.t. $c = ab$.

1:	`UMULL`	c_0, c_1, a_0, b_0	$\{ (c_1, c_0) = a_0 b_0 \}$
2:	`UMULL`	c_2, c_3, a_0, b_1	$\{ (c_3, c_2) = a_0 b_1 \}$
3:	`UMULL`	b_1, a_0, a_1, b_1	$\{ (a_0, b_1) = a_1 b_1 \}$
4:	`UMULL`	b_0, a_1, a_1, b_0	$\{ (a_1, b_0) = a_1 b_0 \}$
5:	`ADDS`	c_1, c_2	$\{ (c_1, \gamma) = c_1 + c_2 \}$
6:	`ADCS`	c_2, c_3, b_1	$\{ (c_2, \gamma) = c_3 + b_1 + \gamma \}$
7:	`ADCS`	$c_3, a_0, 0$	$\{ c_3 = a_0 + \gamma \}$
8:	`ADDS`	c_1, b_0	$\{ (c_1, \gamma) = c_1 + b_0 \}$
9:	`ADCS`	c_2, a_1	$\{ (c_2, \gamma) = c_2 + a_1 + \gamma \}$
10:	`ADCS`	$c_3, c_3, 0$	$\{ c_3 = c_3 + \gamma \}$

Multiplication 32×32-byte multiplications are implemented using a 2-level subtractive Karatsuba, hence requiring nine 8×8-byte multiplications. On ARM Cortex-M4 processors these can be implemented using only 10 instructions, as shown in Listing 1: first, the four partial products $(a_0b_0, a_0b_1, a_1b_1, a_1b_0)$ are computed with four `UMULL` instructions. Then, they are added up using only six `ADDS/ADCS` instructions, as shown in Figure 1. In practice, Listing 1 corresponds to a typical scanning approach with quadratic complexity tailored down for ARM Cortex-M4 processors. However, it does not correspond to the straightforward implementation of neither operand scanning or product scanning techniques, as the partial products are all computed first and then accumulated following an ideal circular route over the 2×2 lattice of partial products. This way of accumulating the partial products allows to quickly ripple the carry values γ over all limbs and minimize the number of additions with carry, thus saving a few instructions and clock cycles.

Squaring Similar to multiplications, 32-byte squaring operations are implemented using a 2-level subtractive Karatsuba, hence requiring nine 8-byte squaring operations. Since $a_0b_1 = a_1b_0$ in case of squaring, one partial product can be skipped and 8-byte squarings can be implemented on ARM Cortex-M4 processors using only 9 instructions, as shown in Listing 2: first, the partial products a_0^2 and a_1^2 are computed. Then, the partial product a_0a_1 is computed and added twice to (c_1, c_2) while rippling the carry through c_3, as shown in Figure 2.

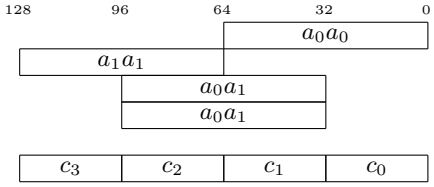

Fig. 2. 8-byte Squaring.

Listing 2 8-byte Squaring.

Input: $a = (a_0, a_1)$.
Output: $c = (c_0, c_1, c_2, c_3)$ s.t. $c = a^2$.

1:			
2:	`UMULL`	c_0, c_1, a_0, a_0	$\{ (c_1, c_0) = a_0^2 \}$
3:	`UMULL`	c_2, c_3, a_1, a_1	$\{ (c_3, c_2) = a_1^2 \}$
4:	`UMULL`	a_0, a_1, a_0, a_1	$\{ (a_1, a_0) = a_0 a_1 \}$
5:	`ADDS`	c_1, c_1, a_0	$\{ (c_1, \gamma) = c_1 + a_0 \}$
6:	`ADCS`	c_2, c_2, a_1	$\{ (c_2, \gamma) = c_2 + a_1 + \gamma \}$
7:	`ADCS`	$c_3, c_3, 0$	$\{ c_3 = c_3 + \gamma \}$
8:	`ADDS`	c_1, c_1, a_2	$\{ (c_1, \gamma) = c_1 + a_2 \}$
9:	`ADCS`	c_2, c_2, a_1	$\{ (c_2, \gamma) = c_2 + a_1 + \gamma \}$
10:	`ADCS`	$c_3, c_3, 0$	$\{ c_3 = c_3 + \gamma \}$

Multiplication by 121666 Since the constant 121666 fits into a 32-bit word (`0x0001DB42`), then the `UMAAL` instruction can be used to compute the multiplications by 121666 with 9 instructions using the following trick: first, one `UMULL` by 121666 is performed, then the constant is decreased by one and seven `UMAAL` instructions with 121665 are performed. The `UMAAL` instruction adds a_1 to the multiplication result $121665a_1$, which yields the same result as $121666a_1$. The final result is stored in $(a_0, \ldots, a_7, \texttt{rHI})$ which can be then directly reduced. In this way, the accumulator does not need to be shifted, hence leading to a very fast and compact implementation (cf. Listing 3).

Listing 3 8-byte Multiplication by 121666.

Input: $a = (a_i)_{0 \le i \le 7}$.
Output: $a = (a_0, \ldots, a_7, \texttt{rHI}) = 121666a$.

1:	`UMULL`	$a_0, \texttt{rHI}, a_0, c$	$\{ (\texttt{rHI}, a_0) = 121666a_0 \}$
2:	`UMAAL`	$a_1, \texttt{rHI}, a_1, c$	$\{ (\texttt{rHI}, a_1) = 121665a_1 + a_1 + \texttt{rHI} \}$
3:	`UMAAL`	$a_2, \texttt{rHI}, a_2, c$	$\{ (\texttt{rHI}, a_2) = 121665a_2 + a_2 + \texttt{rHI} \}$
4:	`UMAAL`	$a_3, \texttt{rHI}, a_3, c$	$\{ (\texttt{rHI}, a_3) = 121665a_3 + a_3 + \texttt{rHI} \}$
5:	`UMAAL`	$a_4, \texttt{rHI}, a_4, c$	$\{ (\texttt{rHI}, a_4) = 121665a_4 + a_4 + \texttt{rHI} \}$
6:	`UMAAL`	$a_5, \texttt{rHI}, a_5, c$	$\{ (\texttt{rHI}, a_5) = 121665a_5 + a_5 + \texttt{rHI} \}$
7:	`UMAAL`	$a_6, \texttt{rHI}, a_6, c$	$\{ (\texttt{rHI}, a_6) = 121665a_6 + a_6 + \texttt{rHI} \}$
8:	`UMAAL`	$a_7, \texttt{rHI}, a_7, c$	$\{ (\texttt{rHI}, a_7) = 121665a_7 + a_7 + \texttt{rHI} \}$

Table 1. Implementation Results.

Operation	Speed [Cycles]	Code [Bytes]	Stack [Bytes]
Addition	106	138	32
Subtraction	108	148	32
Multiplication by 121666	72	116	24
Multiplication	546	1,264	148
Squaring	362	882	104
Inversion	96,337	484	480
$[k] \cdot \mathbf{P}$ (**M** only)	1,658,083	2,952	740
$[k] \cdot \mathbf{P}$ (**M** and **S**)	1,423,667	3,750	740
$[k] \cdot \mathbf{P}$ (**M** and **S** + rand.)	1,563,582	3,786	744

4 Implementation Results

The software was cross-compiled using the GNU Compiler Collection for ARM Embedded Processors version `4.9.3` release `20150529` with the options `-O2 -mthumb -mcpu=cortex-m4` and tested on a `STM32F411RE` ARM Cortex-M4 using a STM32 Nucleo-64 development board [STM].

The code size was measured summing up the size of the `.text`, `.bss` and `.data` segments, as obtained from the GNU `arm-none-eabi-size`. The number of clock cycles was measured using the internal clock cycle counter (CYCCNT) of the Data Watchpoint and Trace Unit (DWT), as available on ARM Cortex-M4 processors, with the aid of the ARM mbed library [ARMa]. Note that the reported number of clock cycles includes the overheads for calling and returning from the considered function under test. Finally, the stack usage was estimated filling the memory with a canary up to 1024 words and then checking how many words were changed after a call to the considered function under test.

The results are summarized in Table 1, where the performance of Curve25519 scalar multiplication is reported in three versions: (1) without randomization and using multiplications in place of squaring operations ("**M** only"), (2) without randomization and using a dedicated routine for squaring operations ("**M** and **S**"), and (3) *with* randomization and using a dedicated routine for squarings ("**M** and **S** + rand"). Note that all the reported results are constant-time. Also, our implementation does not overwrite the input values $(k, \mathbf{P})$ of the scalar multiplication. In case the input parameters can be overwritten, i.e. $\mathbf{Q} = [k] \cdot \mathbf{P}$ overwrites $\mathbf{P}$, then both speed and code size can be further improved.

Table 2 provides a comparison of our results to existing X25519 implementations on 8-bit, 16-bit, and 32-bit embedded processors. In particular, our full-radix X25519 implementation is $\approx 21\%$ faster and $\approx 9\%$ smaller than the corresponding reduced-radix implementation on ARM Cortex M4 processors [dG15]. Apart from the different radix representation, these two implementations mainly

Table 2. Curve25519 on Embedded Processors.

Platform	Multiply [Cycles]	Square [Cycles]	Curve25519 [Cycles]	[Bytes]
8-bit AVR ATmega [HS13]	6,868	–	22,791,580	–
8-bit AVR ATmega [NLD15]	7,555*	5,666*	20,153,658	–
8-bit AVR ATmega [DHH+15]	4,961	3,324	13,900,397	17,710
16-bit MSP430† [HMH+15]	3,606	–	9,139,739	11,778
16-bit MSP430† [DHH+15]	3,193*	2,426*	7,933,296	13,112
16-bit MSP430‡ [HMH+15]	2,488	–	6,513,011	8,956
16-bit MSP430‡ [DHH+15]	2,079*	1,563*	5,301,792	10,088
32-bit ARM Cortex-M0‡ [DHH+15]	1,294	857	3,589,850	7,900
32-bit ARM Cortex-M4‡ [dG15]	631*	563*	1,816,351	4,140
32-bit ARM Cortex-M4‡ (This Work)	546*	–	**1,658,083**	**2,952**
32-bit ARM Cortex-M4‡ (This Work)	546*	362*	**1,423,667**	**3,750**
32-bit ARM Cortex-M4‡ (This Work)	546*	362*	**1,563,582**◇	**3,786**

* Including (fast) reduction.
† With 16 × 16-bit hardware multiplier.
‡ With 32 × 32-bit hardware multiplier.
◇ Using randomized projective coordinates.

differ in the squaring to multiplication ratio. The former has a ratio of ≈ 0.66, while the latter of ≈ 0.89, thus clarifying the obtained speed results. Note that the authors became aware of the independent results of Wouter de Groot's master thesis [dG15] only after finishing their work.

Acknowledgements This work was partly funded by the German Federal Ministry of Education and Research (BMBF) in the project SIBASE under grant number 01IS13020A.

References

ARMa. ARM®, *Arm® mbed™ library*, http://developer.mbed.org/.
ARMb. ARM®, *ARM® and Thumb®-2 Instruction Set*, http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001m/QRC0001_UAL.pdf.
Ber. Daniel J. Bernstein, *25519 naming*, https://www.ietf.org/mail-archive/web/cfrg/current/msg04996.html.
Ber06. Daniel J Bernstein, *Curve25519: new Diffie-Hellman speed records*, Public Key Cryptography – PKC 2006, Springer, 2006, pp. 207–228.
Cor99. Jean-Sébastien Coron, *Resistance against differential power analysis for elliptic curve cryptosystems*, pp. 292–302, Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

CP06. Richard Crandall and Carl B Pomerance, *Prime numbers: a computational perspective*, vol. 182, Springer, 2006.

dG15. Wouter de Groot, *A performance study of X25519 on Cortex M3 and M4*, https://alexandria.tue.nl/extra2/afstversl/wsk-i/Groot_2015.pdf, http://github.com/weedegee/x25519-cortexm4, September 2015.

DHH+15. Michael Düll, Björn Haase, Gesine Hinterwälder, Michael Hutter, Christof Paar, Ana Helena Sánchez, and Peter Schwabe, *High-speed curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers*, Designs, Codes and Cryptography **77** (2015), no. 2, 493–514.

HMH+15. Gesine Hinterwälder, Amir Moradi, Michael Hutter, Peter Schwabe, and Christof Paar, *Latincrypt 2014*, ch. Full-Size High-Security ECC Implementation on MSP430 Microcontrollers, pp. 31–47, Springer, 2015.

HS13. Michael Hutter and Peter Schwabe, *NaCl on 8-Bit AVR Microcontrollers.*, AFRICACRYPT, Springer, 2013, pp. 156–172.

Mon87. Peter L Montgomery, *Speeding the pollard and elliptic curve methods of factorization*, Mathematics of computation **48** (1987), no. 177, 243–264.

NLD15. Erick Nascimento, Julio López, and Ricardo Dahab, *Efficient and secure elliptic curve cryptography for 8-bit avr microcontrollers*, Security, Privacy, and Applied Cryptography Engineering (Rajat Subhra Chakraborty, Peter Schwabe, and Jon Solworth, eds.), LNCS, vol. 9354, Springer, 2015, pp. 289–309 (English).

STM. STMicroelectronics, *STM32F411RE High-performance access line*, http://www2.st.com/resource/en/datasheet/stm32f411re.pdf.