Efficient Curve25519 Implementation for ARM Microcontrollers

Hayato Fujii[1], **Diego F. Aranha**[1]

[1]Institute of Computing – University of Campinas (Unicamp)
Av. Albert Einstein, 1251 – 13083-852 – Campinas – SP – Brazil

{hayato.fujii,dfaranha}@ic.unicamp.br

Abstract. *This work investigates efficient and secure implementations of Curve25519 to build a key exchange protocol on an ARM Cortex-M4 microcontroller, along with the related signature scheme Ed25519 and a digital signature scheme proposal called qDSA. As result, performance-critical operations, such as modular multiplication, are greatly optimized; in this particular case, a 50% speedup is achieved, impacting the performance of higher-level protocols.*

1. Introduction

The ubiquity of technology tends to pervade every single area of knowledge, including environmental, personal, professional and industrial applications. Small-factor computing devices are deployed to identify an in-manufacturing item in a production line, to detect changes in the chemical properties of the soil where crops are cultivated, and to control life-supporting equipment, such as pacemakers and implantable defibrillators [Atzori et al. 2010]. These devices are equipped with the ability of running private, safety-critical or legally liable activities, sensible data collection, manipulation, and transmission. Complex schemes providing data security are hardly implemented, despite the relevance of collected data. For example, sensor networks may collect personally identifiable information available in tags inside cars for surveillance purposes without any kind of data protection.

A possible way to deploy security in new devices is to reuse well-known building blocks, such as the Transport Layer Security (TLS) protocol. In comparison with reinventing the wheel, using a new, under-analyzed option, this has a major advantage of avoiding risky security decisions that may repeat issues already solved in TLS. In the handshaking phase, asymmetric (or public key) cryptographic schemes are largely used; namely, key exchanges and digital signatures.

In the Request for Comments (RFCs) 7748 and 8032, published by the Internet Engineering Task Force (IETF), two asymmetric cryptographic protocols based on the Curve25519 elliptic curve and its (twisted) Edwards form are recommended and slated for use in the TLS suite: the elliptic curve Diffie-Hellman key exchange using Curve25519 [Bernstein 2006] called X25519 and the Ed25519 digital signature scheme [Bernstein et al. 2012]. These schemes rely on a careful choice of parameters, favoring secure and efficient implementations of finite field and elliptic curve arithmetic with smaller room for mistakes due to their overall implementation simplicity. Protocols based on the curve are used in numerous softwares, including The Tor Project, OpenSSH and the Google Chrome browser.

The main objective of this work was to provide an efficient implementation of the Curve25519-based cryptographic protocols, resistant to side-channel attacks; in particular, timing and cache attacks. This required efficient underlying arithmetic modulo $2^{255} - 19$ with no conditional branches to avoid timing issues, leading to handcrafted implementations in assembly code to avoid compiler interference. At the protocol level, elliptic curve group arithmetic was also optimized; in particular, the scalar multiplication operation.

2. Methodology

Efficiently implementing the arithmetic operations of a cryptosystem is the most fundamental way to improve performance in cryptographic schemes. In order to speed up group operations in Curve25519 and in its Twisted Edwards birrationally equivalent curve, strategies to operate with 256-bit long numbers were designed and implemented to speed up basic arithmetic operations of numbers in finite field modulo $p = 2^{255} - 19$, hereon denoted as $\mathbb{F}_p$.

2.1. Curve25519-based Protocols

Curve25519 [Bernstein et al. 2012] is a curve defined over the prime field $\mathbb{F}_{2^{255}-19}$ represented through the Montgomery model

$$\text{Curve25519:} \quad y^2 = x^3 + Ax^2 + x, \tag{1}$$

compactly described by the small value of the coefficient $A = 486662$. This curve model is ideal for curve-based key exchanges, because it allows the scalar multiplication to be computed using x-coordinates only. This property is explored in the Elliptic Curve Diffie-Hellman X25519 key exchange protocol.

The Edwards-curve Digital Signature Algorithm [Bernstein et al. 2012] (EdDSA) is a signature scheme variant of Schnorr signatures based on elliptic curves represented in the Edwards model. Using a birational equivalence, Curve25519 can be also represented in the twisted Edwards model using full coordinates to allow instantiations of secure signature schemes:

$$\text{edwards25519:} \quad -x^2 + y^2 = 1 - \frac{121655}{121666}x^2y^2. \tag{2}$$

When EdDSA is used with Equation 2, the instance is called as the Ed25519 signature scheme.

A Kummer variety $\mathcal{K}$ computed by $\mathcal{K} = E/\langle \pm 1 \rangle$ is foundation of the Quotient Digital Signature Scheme (qDSA) [Renes and Smith 2017]. If E is an elliptic curve, $\mathcal{K}$ turns out to be an one-dimensional space known as the x-line, in which scalar multiplications are still possible. In a instance of qDSA using the parameter $E =$ Curve25519, this scheme allow that X25519 keys be used, without modifications, to sign data.

2.2. Target Architecture

The ARMv7E-M instruction set, present on the Cortex-M4 and higher-end cores, comprises of standard instructions for basic arithmetic (such as addition and addition with carry) and logic operations, but differently from other lower processors classes, there is support for the so-called DSP instructions, which include *multiply-and-accumulate* (MAC) instructions:

- *Unsigned MULtiply Long:* `UMULL rLO, rHI, a, b` takes two unsigned integer words a and b and multiplies them; this double-word result is stored as (`rHI, rLO`).
- *Unsigned MULtiply Accumulate Long:* `UMLAL rLO, rHI, a, b` takes unsigned integer words a and b and multiplies them; the product is added and written back to the double word integer stored as (`rHI, rLO`).
- *Unsigned Multiply Accumulate Accumulate Long:* `UMAAL rLO, rHI, a, b` takes unsigned integer words a and b and multiplies them; the product is added with the word-sized integer stored in `rLO` then added with the word-sized integer `rHI`. The double word integer is then stored as (`rHI, rLO`).

The mentioned MAC instructions takes one CPU cycle for execution in the Cortex-M4 and above [ARM 2010]. However, those instructions deterministically take an 3rd extra cycle to write `rLO` back and a 4th cycle to write into `rHI`. This makes proper instruction scheduling necessary to avoid pipeline stalls.

2.3. $\mathbb{F}_{2^{255}-19}$ Arithmetic Implementation

Each 255-bit integer field element is densely represented, using 2^{32}-radix, implying in eight "limbs" of 32 bits, each one in a little-endian format.

The 256-bit addition is implemented by respectively adding each limb in a lower to higher element fashion. With no extra bits to store in the dense representation, the carry flag, present in the ARM status register, is used to ripple the carry across the limbs without overhead, with the Add-with-Carry (`ADC`) instruction handling it. The result must be always less than $2^{256} - 1$, fitting in the 8 32-bit long limbs, requiring further handling of the carry flag if it is set by the end of the routine. Subtraction follows a similar strategy.

A "weak" modular reduction modulo $2^{256} - 38$ is performed at the end of every field operation in order to avoid extra carry computations between operations, as suggested in [Düll et al. 2015]; this reduction must find a integer less than 2^{256} that is congruent modulo $2^{255} - 19$. "Strong" modular reduction modulo $p = 2^{255} - 19$ is only used when results must be communicated to the other party. This operation is not used in-between operations because the "weak" modular reduction, in the cases after multiplication and squaring for example, takes approximately 10% less cycles than a strong modular reduction modulo p.

The multiplication by a word operation is used a single time when doubling a point in X25519, where a 256-bit long integer must be multiplied by $d = 121666$. This operation follows the algorithm described in [Santis and Sigl 2016].

2.3.1. Multiplying two 256-bit numbers

The $256 \times 256 \rightarrow 512$-bit multiplication follows a product-scanning like approach; more specifically, Full Operand-Caching [Seo and Kim 2015], using parameters $n = 8$, $e = 3$, $r = \lfloor n/e \rfloor = 2$; since $\lfloor 3/2 \rfloor < 8 - 2 \cdot 3 \leq 3$. Figure 1 illustrates this instance.

Storing partial products in extra registers without adding them avoids potential carry values, which requires at least 2-3 extra CPU cycles to handle. In a trivial implementation, a register accumulator may be used to add the partial values, potentially generating carries. The `UMAAL` instruction can be employed to perform such addition, while also taking

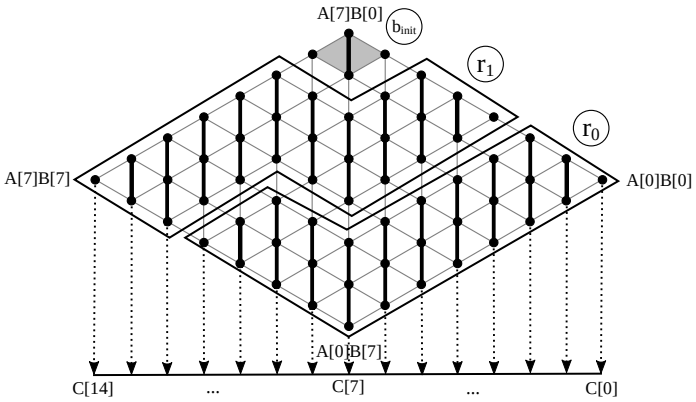

Figure 1. Full Operand-Caching method to multiply two 8-word integers. Black dots represents 32×32-bit multiplications

advantage of the multiplication part to further calculate more partial products. This instruction never generates a carry bit, since $(2^n-1)^2+2(2^n-1) = (2^{2n}-1)$, eliminating the need for extra handling. Partial products generated by this instruction can be forwarded to the next multiply-accumulate(-accumulate) operation; this goes on until all rows are processed. Figure 2 show a toy example of multiplication employing the product scanning approach with 3-word sized operands A and B using the `UMLAL` and `UMAAL` instructions.

The multiplication strategy using the `UMAAL` instruction can be also used to implement a 256-bit multiplication using a operand-scanning approach. Figure 3 shows a toy example of the operand-scanning approach for multiplication with 3-word sized operands A and B using the aforementioned strategy.

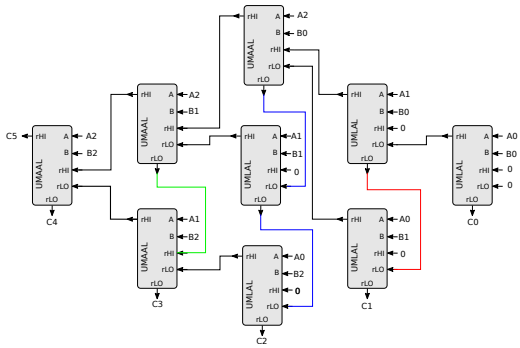

Figure 2. Multiplying two 3-word integers with the product scanning approach using the `UMLAL` and `UMAAL` instructions.

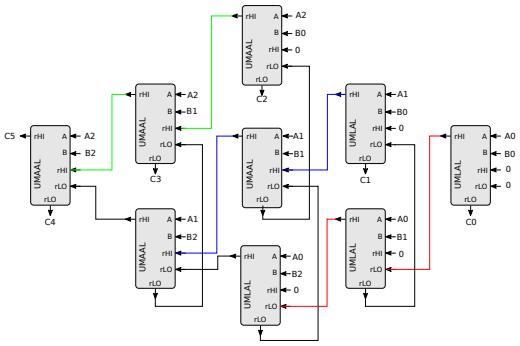

Figure 3. Multiplying two 3-word integers with the operand scanning approach using the `UMLAL` and `UMAAL` instructions.

2.3.2. Squaring

We implemented a squaring operation based on the *Sliding Block Doubling* (SBD) algorithm presented in [Seo et al. 2013]. With the usage of carry flag present in the ARM architecture, both *Sliding Block Doubling* and the *Bottom Line* steps of SBD can be efficiently computed. In order to avoid extra memory access, those two routines may be implemented without reloading operands; because of the need of the carry bit in both those

operations, high register pressure may arise in order to save them into registers. Calculating some multiplications akin to the Initial Block step, as in the Operand Caching [Hutter and Wenger 2011] multiplication method, reduces register usage by spilling partial results in memory. This allows proper carry catching and handling in exchange for a few memory accesses. This method is exemplified in Figure 4.

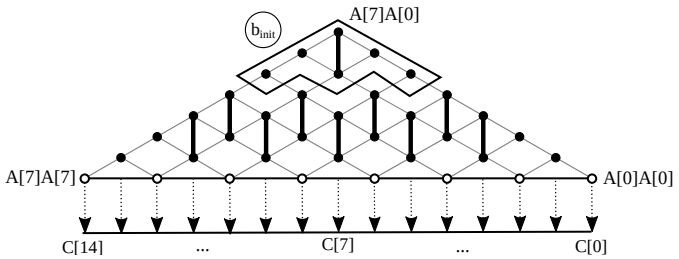

Figure 4. Sliding Block Doubling, computing an initial block beforehand. Black dots represents 32×32-bit multiplications; hollowed ones represent squarings.

3. Results

Table 1 presents timings and Table 2 shows the code size for field operations with implementation described in Section 2.3. In comparison to the previous state-of-art [Santis and Sigl 2016], our addition/subtraction take 18% less cycles; the 256-bit multiplier with a weak reduction is almost 50% faster and the squaring operation takes 30% less cycles. Implementation of all arithmetic operations take less code space in comparison to [Santis and Sigl 2016], ranging from 20% savings in the addition to 50% in the 256-bit multiplier.

Table 1. Timings in cycles for arithmetic in $\mathbb{F}_{2^{255}-19}$ executing in ARM Cortex-M4 controllers; average of 256 executions. Two 256-multiplication implementations are measured: one based on the Fully Consecutive Operand Caching ("COC") and the other one based on Operand Scanning ("OpScan"). [a] Teensy 3.1 board. [b] STM32F401C board.

	Cortex	Add/Sub	Mult		Mult by word	Square	Inversion
[Santis and Sigl 2016]	M4	106	546		**72**	362	96337
			COC	**OpScan**			
This work	M4 @ 48 MHz [a]	**91/89**	**284**	329	95	**251**	**66681**
	M4 @ 72 MHz [a]	**91/89**	**311**	358	100	**290**	**77356**
	M4 @ 84 MHz [b]	**91/89**	**274**	321	92	**245**	**64955**

As noted by [Haase 2017], cycle counts of a given code running on the same Cortex-M4-based controller can be different depending on the clock frequency set on the controller. This may happen when a slower frequency is set to the memory controller, causing stalls on the CPU if the running code depends on memory access. Operations relying on memory operations, such as the 256-bit multiplication and squaring, use 10% more cycles when the controller is set to a 33% higher frequency, for example. This behavior is also present on cryptographic schemes implementations, since those are subject to compiler interference once those were implemented in a higher level language.

Cycle counts of the X25519 function, Ed25519 and qDSA (instantiated with Curve25519) operations implementations executed on the target processors are shown in Tables 3 and 4.

Table 2. Code size in bytes for implementing arithmetic in $\mathbb{F}_{2^{255}-19}$, X25519, Ed25519 and qDSA with Curve25519 protocols on the Cortex-M4. Code size for protocols considers the entire software stack needed to perform the specific action, including but not limited to field operations, hashing, tables for scalar multiplication and other algorithms.

	Add	**Sub**	**Mult**	**Mult by word**	**Square**
De Santis [Santis and Sigl 2016]	138	148	1264	116	882
This work	110	108	**622**	**92**	**562**
	Inversion	**X25519**	**Ed25519 Key Gen.**	**Ed25519 Sign**	**Ed25519 Verify**
De Santis [Santis and Sigl 2016]	484	**3786**	-	-	-
This work	**328**	4152	**21265**	**22162**	**28240**
	qDSA Key Gen.	**qDSA Sign**	**qDSA Verify**		
Left to Right Montgomery	14546	20720	15856		
Right to Left Montgomery [Oliveira et al. 2017]	24762	29756	25516		

X25519 was implemented using the standard Montgomery ladder over the x-coordinate. Standard tricks like randomized projective coordinates (amounting to 1% performance penalty) and constant-time conditional swaps were implemented for protection against power analysis and timing attacks, respectively. Our implementation is 42% faster than De Santis and Sigl [Santis and Sigl 2016] while staying competitive in terms of code size.

Key generation and message signing using the Ed25519 scheme requires a fixed-point scalar multiplication, here implemented using the signed-comb algorithm proposed in [Hamburg 2012] running in constant-time. We use five teeth for each of the five blocks and 10 combs for each of the 5 blocks (11 combs for the last one) of the signed-binary representation of the scalar, much like in the multi-comb approach described in [Hankerson et al. 2003]. Those parameters where chosen due to the performance balance between a direct and linear table scan to access precomputed data; this one required if the presence of cache memory on the CPU requires protection against side-channel attacks. To compute the scalar multiplication, our implementation requires 50 point additions and 254 point doublings. Five lookup tables of 16 points in Extended Projective coordinate format with $z = 1$ are used, adding up to approximately 7.5 KiB of data. Verification requires a double-point multiplication involving the generator B and point A using a w-NAF interleaving technique [Hankerson et al. 2003], with a window of width 5 for the A point, generating on-the-fly an auxiliary table in volatile memory using 3 KiB. The group generator B is interleaved using a window of width 7, implying in a lookup table of 32 points stored in Extended Projective coordinate format with $z = 1$ taking 3 KiB of ROM.

The qDSA signature scheme is also impacted by the fixed-base scalar multiplication. A right-to-left implementation of the Montgomery ladder [Oliveira et al. 2017] with a 8 KiB table is used to store multiples of the generator point in ROM. This approach was chosen over methods requiring auxiliary tables to compute a fixed-base scalar multiplication. In these cases, protection to table accesses in cases where cache memory is present may cause extra performance overhead proportional to the number of entries of a precomputed table.

Our X25519 implementation is inherently protected against timing attacks (i. e. constant-time execution); the Ed25519 implementation also runs in constant-time (in exception to the verification procedure) and inherently protected against cache attacks due to lack of cache memory on tested platforms. The qDSA implementation is also inherently constant time, and, since no secret indexes are used to access the auxiliary table, there's no

need to protect it against cache attacks.

Table 3. Timings in 10^3 cycles to compute X25519; and key generation, signature and verification of a 5-byte message in the Ed25519 scheme. Numbers are an average of 256 executions. The measures for this work takes in account the best 256-bit multiplier shown on Table 1. [a] Teensy 3.1 board. [b] STM32F401C board.

	Cortex	X25519	Ed25519 Key Gen.	Ed25519 Sign	Ed25519 Verify
De Santis [Santis and Sigl 2016]	M4	1,563.8	-	-	-
This work	M4 @ 48 MHz [a]	**925.7**	**353.0**	**501.7**	**1,323**
	M4 @ 72 MHz [a]	**1,036.6**	**385.6**	**537.2**	**1,463.6**
	M4 @ 84 MHz [b]	**913.8**	**394.7**	**549.0**	**1,360.8**
[Moon 2012]	M4 @ 48 MHz [a]	-	693.9	750.5	1,967.7
	M4 @ 72 MHz [a]	-	738.4	796.7	2,026.4

Table 4. Timings in 10^3 cycles for key generation, signature and verification of a 5-byte message in the qDSA scheme. Numbers taken as the average of 256 executions on a Teensy 3.1 board clocked at 48 MHz.

	qDSA Key Gen.	qDSA Sign	qDSA Verify
Left to Right Montgomery	927.9	1,059.1	1,746.2
Right to Left Montgomery [Oliveira et al. 2017]	614.5	744.8	1,451.8

4. Conclusion

Given the performance numbers shown in Tables 1, 3, and 4, we consider that our implementation is competitive in comparison to the previous state-of-art while not using too much ROM (Table 2). Using Curve25519 and its corresponding Twisted Edwards form in well-known protocols is beneficial in terms of security, mostly due to its maturity and its widespread usage to the point of becoming a *de facto* standard.

Partial results of this work were published in the Fifth International Conference on Cryptology and Information Security in Latin America (Latincrypt 2017) in the paper "Curve25519 for the Cortex-M4 and Beyond" [Fujii and Aranha 2017] and in the Seventh International Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE 2017) [Faz-Hernández et al. 2017].

The full dissertation, as in its preprint version, is avaliable at `http://ic.unicamp.br/~ra180790/msc/`.

Acknowledgments. The authors acknowledge financial support from LG Electronics Inc. during the development of this work, under the project "*Efficient and Secure Cryptography for IoT*".

References

ARM (2010). Cortex-M4 Devices Generic User Guide. Avaliable on `http://infocenter.arm.com/help/index.jsp?topic=%2Fcom.arm.doc.dui0553a%2FCHDBFFDB.html`.

Atzori, L., Iera, A., and Morabito, G. (2010). The internet of things: A survey. *Computer Networks*, 54(15):2787–2805.

Bernstein, D. J. (2006). Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer.

Bernstein, D. J., Duif, N., Lange, T., Schwabe, P., and Yang, B. (2012). High-speed high-security signatures. *J. Cryptographic Engineering*, 2(2):77–89.

Düll, M., Haase, B., Hinterwälder, G., Hutter, M., Paar, C., Sánchez, A. H., and Schwabe, P. (2015). High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Des. Codes Cryptography*, 77(2-3):493–514.

Faz-Hernández, A., Fujii, H., Aranha, D. F., and López, J. (2017). A secure and efficient implementation of the quotient digital signature algorithm (qdsa). In *SPACE*, volume 10662 of *Lecture Notes in Computer Science*, pages 170–189. Springer.

Fujii, H. and Aranha, D. F. (2017). Curve25519 for the Cortex-M4 and Beyond. In *Progress in Cryptology – LATINCRYPT 2017: 5th International Conference on Cryptology and Information Security in Latin America 2017, Proceedings*, Lecture Notes in Computer Science. Springer. (to appear).

Haase, B. (2017). Memory bandwidth influence makes Cortex M4 benchmarking difficult. Avaliable on `https://ches.2017.rump.cr.yp.to/fe534b32e52fcacee026786ff44235f0.pdf`.

Hamburg, M. (2012). Fast and compact elliptic-curve cryptography. Avaliable on `https://eprint.iacr.org/2012/309.pdf`.

Hankerson, D., Menezes, A. J., and Vanstone, S. (2003). *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Hutter, M. and Wenger, E. (2011). Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In *CHES*, volume 6917 of *Lecture Notes in Computer Science*, pages 459–474. Springer.

Moon, A. (2012). Implementations of a fast Elliptic-curve Digital Signature Algorithm. Available at `https://github.com/floodyberry/ed25519-donna`.

Oliveira, T., López, J., Hisil, H., Faz-Hernández, A., and Rodríguez-Henríquez, F. (2017). How to (pre-)compute a ladder - improving the performance of X25519 and X448. In *SAC*, volume 10719 of *Lecture Notes in Computer Science*, pages 172–191. Springer.

Renes, J. and Smith, B. (2017). qDSA: Small and Secure Digital Signatures with Curve-Based Diffie-Hellman Key Pairs. In *ASIACRYPT (2)*, volume 10625 of *Lecture Notes in Computer Science*, pages 273–302. Springer.

Santis, F. D. and Sigl, G. (2016). Towards Side-Channel Protected X25519 on ARM Cortex-M4 Processors. In *SPEED-B*, Utrecht, The Netherlands.

Seo, H. and Kim, H. (2015). Consecutive operand-caching method for multiprecision multiplication, revisited. *J. Inform. and Commun. Convergence Engineering*, 13(1):27–35.

Seo, H., Liu, Z., Choi, J., and Kim, H. (2013). Multi-precision squaring for public-key cryptography on embedded microprocessors. In *INDOCRYPT*, volume 8250 of *Lecture Notes in Computer Science*, pages 227–243. Springer.