Curve25519 for the Cortex-M4 and Beyond

Hayato Fujii(✉) and Diego F. Aranha

Institute of Computing, University of Campinas, Campinas, Brazil
hayato@lasca.ic.unicamp.br, dfaranha@ic.unicamp.br

Abstract. We present techniques for the implementation of a key exchange protocol and digital signature scheme based on the Curve25519 elliptic curve and its Edwards form, respectively, in resource-constrained ARM devices. A possible application of this work consists of TLS deployments in the ARM Cortex-M family of processors and beyond. These devices are located towards the lower to mid-end spectrum of ARM cores, and are typically used on embedded devices. Our implementations improve the state-of-the-art substantially by making use of novel implementation techniques and features specific to the target platforms.

Keywords: ECC · Curve25519 · X25519 · Ed25519 · ARM Cortex-M

1 Introduction

The growing number of devices connected to the Internet collecting and storing sensitive information raises concerns about the security of their communications and of the devices themselves. Many of them are equipped with microcontrollers constrained in terms of computing or storage capabilities, and lack tamper resistance mechanisms or any form of physical protection. Their attack surface is widely open, ranging from physical exposure to attackers and ease of access through remote availability. While designing and developing efficient and secure implementations of cryptography is not a new problem and has been an active area of research since at least the birth of public-key cryptography, the application scenarios for these new devices imposes new challenges to cryptographic engineering.

A possible way to deploy security in new devices is to reuse well-known and well-analyzed building blocks, such as the Transport Layer Security (TLS) protocol. In comparison with reinventing the wheel using a new and possibly proprietary solution, this has a major advantage of avoiding risky security decisions that may repeat issues already solved in TLS. In RFC 7748 and RFC 8032, published by the Internet Engineering Task Force (IETF), two cryptographic protocols based on the Curve25519 elliptic curve and its Edwards form are recommended and slated for future use in the TLS suite: the Diffie-Hellman key exchange using Curve25519 [2] called X25519 [3] and the Ed25519 digital signature scheme [5]. These schemes rely on a careful choice of parameters, favoring

T. Lange and O. Dunkelman (Eds.): LATINCRYPT 2017, LNCS 11368, pp. 109–127, 2019.
https://doi.org/10.1007/978-3-030-25283-0_6

secure and efficient implementations of finite field and elliptic curve arithmetic with smaller room for mistakes due to their overall implementation simplicity.

Special attention must be given to side-channel attacks, in which operational aspects of the implementation of a cryptographic algorithm may leak internal state information that allows an attacker to retrieve secret information. Secrets may leak through the communication channel itself, power consumption, execution time or radiation measurements. Information leaked through cache latency or execution time already allows powerful timing attacks against naive implementations of symmetric and public-key cryptography [20]. More intrusive attacks also attempt to inject faults at precise execution times, in hope of corrupting execution state to reveal secret information [8]. Optimizing such implementations to achieve an ideal balance between resource efficiency and side-channel resistance further complicates matters, beckoning algorithmic advances and novel implementation strategies.

This work presents techniques for efficient, compact and secure implementation against timing and caching attacks of both X25519 and Ed25519 algorithms, with an eye towards possible application for TLS deployments on constrained ARM processors[1]. Our main target platform is the Cortex-M family of microcontrollers starting from the Cortex-M4, but the same techniques can be used in higher-end CPUs such as the ARM Cortex-A series.

Contributions. We first present an ARM-optimized implementation of the finite field arithmetic modulo the prime $p = 2^{255} - 19$. The main contribution in terms of novelty is an efficient multiplier largely employing the powerful multiply-and-accumulate DSP instructions in the target platform. The multiplier uses the full 32-bit width and allows to remove any explicit addition instructions to accumulate results or propagate carries, as all of these operations are combined with the DSP instructions. These instructions are not present in the Cortex-M0 microcontrollers, and present a variable-time execution in the Cortex-M3 [13], hence the choice of focusing out efforts on the Cortex-M4 and beyond. The same strategy used for multiplication is adapted to squarings, with similar success in performance. Following related work [11], intermediate results are reduced modulo $2p$ and ultimately reduced modulo p at the end of computation.

Finite field arithmetic is then used to implement the higher levels, including group arithmetic and cryptographic protocols. The key agreement implementation uses homogeneous projective coordinates in the Montgomery curve in order to take advantage of the constant-time Montgomery ladder as the scalar multiplication algorithm, protecting against timing attacks. The digital signature scheme implementation represents points on a Twisted Edwards curve using projective extended coordinates, benefiting of efficient and unified point addition and doubling. The most time-consuming operation in the scheme is the fixed-point scalar multiplication, which uses the signed comb method as introduced by Hamburg [15], using approximately 7.5 KiB of precomputed data and running approximately two times faster in comparison to the Montgomery ladder. Side-channel security is achieved using isochronous (constant time) code

[1] https://www.keil.com/pack/doc/mw/Network/html/use_mbed_tls.html.

execution and linear table scan countermeasures. We also evaluate a different way to implement the conditional selection operation in terms of their potential resistance against profiled power attacks [25]. Experiments conducted on a Cortex-M4 development board indicate that our work provides the fastest implementations of these specific algorithms for our main target architecture.

Organization. Section 2 briefly describes features of our target platform. Section 3 documents related works in this area, by summarizing previous implementations of elliptic curve algorithms in ARM microcontrollers. In Sect. 4 discusses our techniques for finite field arithmetic in detail, focusing in the squaring and multiplication operations. Section 5 describes the algorithms using elliptic curve arithmetic in the key exchange and digital signatures scenarios. Finally, Sect. 6 presents experimental results and implementation details.

Code Avaliability. For reproducibility, the prime field multiplication code is publicly available at https://github.com/hayatofujii/curve25519-cortex-m4.

2 ARMv7 Architecture

The ARMv7 architecture is a reduced instruction set computer (RISC) using a load-store architecture. Processors with this technology are equipped with 16 registers: 13 general purpose, one as the program counter (`pc`), one as the stack pointer (`sp`), and the last one as the link register (`lr`). The latter can be freed up by saving it in slower memory and retrieving it after the register has been used.

The processor core has a three-stage pipeline which can be used to optimize batch memory operations. Memory access involving n registers in these processors takes $n + 1$ cycles if there are no dependencies (for example, when a loaded register is the address for a consecutive store). This can happen either in a sequence of loads and stores or during the execution of instructions involving multiple registers simultaneously.

The ARMv7E-M instruction set is also comprised of standard instructions for basic arithmetic (such as addition and addition with carry) and logic operations, but differently from other lower processors classes, the Cortex-M4 has 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; the upper half result is written back to `rHI` and the lower half is written into `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 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 again with the word-sized integer `rHI`. This double-word integer is then written back into `rLO` and `rHI`, respectively the lower half and the upper half of the result.

ARM's Technical Reference Manual of the Cortex-M4 core [1] states that all the mentioned MAC instructions take one CPU cycle for execution in the Cortex-M4 and above. However, those instructions deterministically take an extra three cycles to write the lower half of the double-word result, and a final extra cycle to write the upper half. Therefore, proper instruction scheduling is necessary in order to avoid pipeline stalls and to make best use of the delay slots.

The ARM Cortex-A cores are computationally more powerful than their Cortex-M counterparts. Cortex-A based processors can run robust operating systems due to extra auxiliary hardware; additionally, they may have a NEON engine, which is a Single Instruction-Multiple Data (SIMD) unit. Aside from that, those processors may have sophisticated out-of-order execution and extra pipeline stages.

3 Related Work

Research in curve-based cryptography proceeds in several directions: looking for efficient elliptic curve parameters, instantiating and implementing the respective cryptographic protocols, and finding new applications. More recently, isogeny-based cryptography [18], which uses elliptic curves, was proposed as candidates for post-quantum cryptography.

3.1 Scalar Multiplication

Düll *et al.* [11] implemented X25519 and its underlying field arithmetic on a Cortex-M0 processor, equipped with a $32 \times 32 \rightarrow 32$-bit multiplier. Since this instruction only returns the lower part of the product, this multiplier is abstracted as a smaller one ($16 \times 16 \rightarrow 32$) to facilitate a 3-level Refined Karatsuba implementation, taking 1294 cycles to complete on the same processor. Their 256-bit squaring uses the same multiplier strategy with standard tricks to save up repeated operations, taking 857 cycles. Putting all together, an entire X25519 operation takes about 3.6 M cycles with approximately 8 KiB of code size.

On the Cortex-A family of processor cores, implementers may use NEON, a SIMD instruction set executed in its own unit inside the processor. Bernstein and Schwabe [7] reported 527,102 Cortex-A8 cycles for the X25519 function. In the elliptic curves formulae used in their work, most of the multiplications can be handled in a parallel way, taking advantage of NEON's vector instructions and Curve25519's parallelization opportunities.

The authors are not aware of an Ed25519 implementation specifically targeting the Cortex-M4 core. However, Bernstein's work using Cortex-A8's NEON unit reports 368,212 cycles to sign a short message and 650,102 cycles to verify its validity. The authors point out that 50 and 25 thousand cycles of signing and verification are spent by SUPERCOP-choosen SHA-512 implementation, with room for further improvements.

Four$\mathbb{Q}$ is an elliptic curve providing about 128 bits of security equipped with the endomorphisms ψ and ϕ, providing efficient scalar multiplication [9]. Implementations of key exchange over this elliptic curve in different software and hardware platforms show a factor-2 speedup in comparison to Curve25519 and factor-5 speedup in comparison to NIST's P-256 curve [22]. Liu *et al.* reported [22] a 559,200 cycle count on an ARM Cortex-M4 based processor of their 32-bit implementation of the Diffie-Hellman Key Exchange in this curve.

Generating keys and Schnorr-like signatures over Four$\mathbb{Q}$ takes about 796,000 cycles on a Cortex-M4 based processor, while verification takes about 733,000 cycles on the same CPU [22]. Key generation and signing are aided by a 80-point table taking 7.5KiB of ROM, and verification is assisted by a 256-point table, using 24 KiB of memory. Quotient DSA (qDSA) [27] is a novel signature scheme relying on Kummer arithmetic in order use the same key for DH and digital signature schemes. It relies only on the x-coordinate with the goal of reducing stack usage and the use of optimized formulae for group operations. When instantiated with Curve25519, it takes about 3 million cycles to sign a message and 5.7 million cycles to verify it in a Cortex-M0. This scheme does not rely on an additional table for speedups since low code size is an objective given the target architecture, although this can be done using the ideas from [26] with increased ROM usage.

3.2 Modular Multiplication

Field multiplication is usually the most performance-critical operation, because other non-trivial field operations, such as inversion, are avoided by algorithmic techniques. Multiprecision multiplication algorithms can be ranked on how many single word multiplications are performed. For example, operand scanning takes $O(n^2)$ multiplications, where n is the number of words. Product scanning takes the same number of word multiplications, but reduces the number of memory access by accumulating intermediate results in registers. One of the most popular algorithms that asymptotically reduces such complexity is the Karatsuba multiplication, which takes the computational cost down to $O(n^{\log_2 3})$. This algorithm performs three multiplications of, usually, half-sized operands, thus giving it a divide-and-conquer structure and lowering its asymptotic complexity. As an example of such an application, De Santis and Sigl [28] X25519 implementation on the Cortex-M4 features a two-level Karatsuba multiplier implementation, splitting a 256-bit multiplier down to 64-bit multiplications, each one taking four hardware-supported $32 \times 32 \rightarrow 64$ multiplication instructions.

Memory accesses can be accounted for part of the time consumed by the multiplication routine. Thus, algorithms and instruction scheduling methods which minimize those memory operations are highly desirable, specially on not-so-powerful processors with slow memory access. This problem can be addressed by scheduling the basic multiplications following the product scanning strategy, which can be seen as a rhombus-like structure. However, following this scheme in its traditional way requires multiple stores and loads from memory, since the number of registers available may be not sufficient to hold the full operands. Improvements to reduce the amount of memory operations are present

in the literature: namely, Operand Caching due to Hutter and Wegner [17], further improved by the Consecutive Operand Caching [30] and the Full Operand Caching, both due to Seo *et al.* [29].

The Operand Caching technique reduces the number of memory accesses in comparison to the standard product-scanning by caching data in registers and storing part of the operands in memory. This method resembles the product scanning approach, but instead of calculating a word in its entirety, rows are introduced to compute partial sub-products from each column. This method is illustrated in Fig. 1.

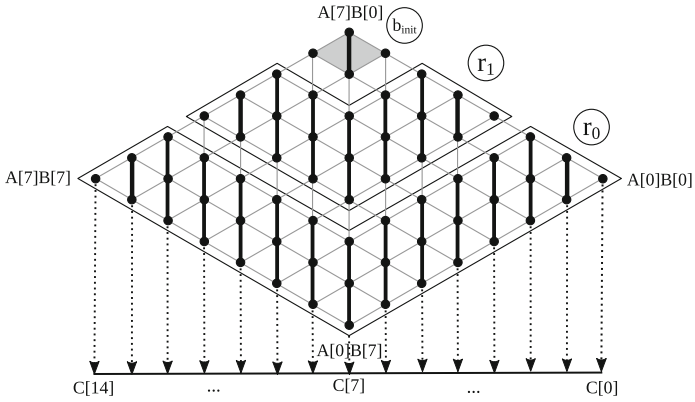

Fig. 1. Operand Caching. Each dot in the rhombus represents one-word multiplication; each column, at the end of its evaluation, is a (partial) word of the product. Vertical lines represents additions.

This method divides product scanning in two steps:

- **Initial Block.** The first step loads part of the operands, and proceeds to calculate the upper part of the rhombus using classical product-scanning.
- **Rows.** In the rightmost part, most of the necessary operands are already loaded from previous calculations, requiring only some extra, low-count operand loads, depending on row width. Product scanning is done until the row ends. Note that, at the end of each column, parts of the operands are previously loaded, hence a small quantity of loads is necessary to evaluate the next column.

At every row change, new operands needs to be reloaded, since the current operands in the registers are not useful at the start of the new row. Consecutive Operand Caching avoids those memory access by rearranging the rows and further improving the quantity of operands already in registers. This algorithm is depicted in Fig. 2.

Note that during the transition between the bottommost row and the one above, part of the operands are already available in registers, solving the reload

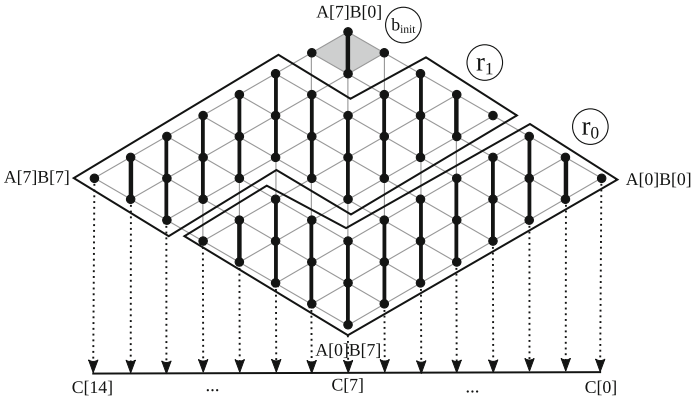

Fig. 2. Consecutive Operand Caching

problem between row changes. Let n be the number of "limbs", r the number of available working registers and number of rows e. Full Operand Caching further improves the quantity of memory access in two cases: if $n - re < e$, the Full Operand Caching structure looks like the original Operand Caching, but with a different multiplication order. Otherwise, Consecutive Operand Caching bottom row's length is adjusted in order to make full use of all available registers at the next row's processing.

3.3 Modular Squaring

The squaring routine can be built by repeating the operands and using the multiplication procedure, saving ROM space. Alternatively, writing a specialized procedure can save cycles by duplicating partial products [23]. The squaring implementation in [28] follows this strategy, specializing the 64-bit multiplication routine to 8 cycles, down from 10. Partial products are calculated then added twice to the accumulator and the resulting carry is rippled away.

Following the product scanning method, Seo's Sliding Block Doubling [30] halves the rhombus structure, allowing to use more registers to store part of the operands and doubling partial products. The algorithm is illustrated in Fig. 3 and can be divided in three parts:

- **Partial Products of the Upper Part Triangle**: an adaption of product scanning calculates partial products (represented by the black dots at the superior part of the rhombus in Fig. 3) and saves them to memory.
- **Sliding Block Doubling of Partial Products**: Each result of the column is doubled by left shifting each result by one, effectively duplicating the partial products. This process must be done in parts because the number of available registers is limited, since they hold parts of the operand.
- **Remaining Partial Products of the Bottom Line**: The bottom line multiplications are squares of part of the operand. These products must be added to their respective partial result of its above column.

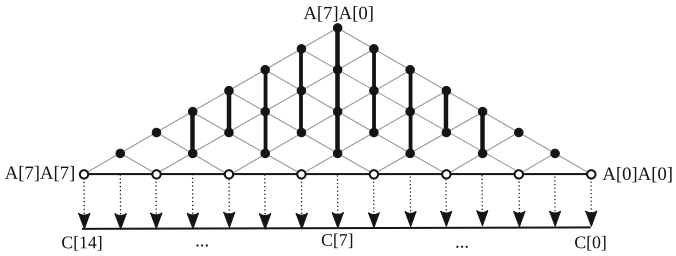

Fig. 3. Sliding Block Doubling. Black dots represent multiplications and white dots represent squarings.

4 Implementation of $\mathbb{F}_{2^{255}-19}$ Arithmetic

Our implementation aims for efficiency, so specific ARM Assembly is thoroughly used and code size is moderately sacrificed for speed. Code portability is a non-goal, so each 255-bit integer field element is densely represented, using 2^{32}-radix, implying in eight "limbs" of 32 bits, each one are in a little-endian format. This contrasts with the reference implementation [2], which use 25 or 26 bits in 32-bits words, allowing carry values requiring proper handling at the expense of more cycles.

Modular Reduction. We call as "weak" reduction a reduction modulo $2^{256} - 38$, performed at the end of every field operation in order to avoid extra carry computations between operations, as in [11]; this reduction finds a integer lesser then 2^{256} that is congruent modulo $2^{255} - 19$. When necessary, a "strong" reduction modulo $2^{255} - 19$ is performed, much like when data must be sent over the wire. This strategy is justified over the extra 10% difference between the "strong" and the "weak" reduction.

Addition and Subtraction. 256-bit addition is implemented by respectively adding each limb in a lower to higher element fashion. The carry flag, present in the ARM status registers, is used to ripple the carry across the limbs. In order to avoid extra bits generated by the final sum, the result is weakly reduced. Subtraction follows a similar strategy.

Multiplication by a 32-bit Word. Multiplication by a single word follows the algorithm described in [28], used to multiply a long integer by 121666, operation required to double a point on Curve25519.

Inversion. This operation follows the standard Itoh-Tsujii addition-chain approach to compute $a^{p-2} \equiv a^{-1} \pmod{p}$, using 11 multiplications and 254 field squarings as proposed in [2]. Adding up the costs, inversion turns to be the most expensive field operation in our implementation.

4.1 Multiplication

The $256 \times 256 \rightarrow 512$-bit multiplication follows a product-scanning like approach; more specifically, Full Operand-Caching. As mentioned in Sect. 3, parameters for

this implementation are $n = 8$, $e = 3$, $r = \lfloor n/e \rfloor = 2$; since $\lfloor 3/2 \rfloor < 8 - 2 \cdot 3 \leq 3$, so Full Operand Caching with a Consecutive-like structure yields the best option.

Catching the Carry Bit. Using product scanning to calculate partial products with a double-word multiplier implies adding partial products of the next column, which in turn might generate carries. A partial column, divided in rows in a manner as described in Operand Caching, can be calculated using Algorithm 1; an example of implementation in ARM Assembly is shown in Listing 1.1. Notation follows as $(\varepsilon, z) \leftarrow w$ meaning $z \leftarrow w \mod 2^W$ and $\varepsilon \leftarrow 0$ if $w \in [0, 2^W]$, otherwise $\varepsilon \leftarrow 1$, where W is the bit-size of a word; (AB) denotes a $2W$-bit word obtained by concatenating the W-bit words A and B.

Algorithm 1. Column computation in product scanning.

Input: Operands A, B; column index k; partial product R_k (calculated during column $k-1$); accumulated carry R_{k+1} (generated from sum of partial products).

Output: (Partial) product $AB[k]$; sum R_{k+1} (higher half part of the partial product for column $k+1$); accumulated carry R_{k+2} (generated from sum of partial products).

$R_{k+2} \leftarrow 0$
for all $(i, j) \mid i + j = k, 0 \leq i < j \leq n - 1$ **do**
  $T \leftarrow 0$
  $(TR_k) \leftarrow A[i] \times B[j] + (T, R_k)$
  $(\varepsilon, R_{k+1}) \leftarrow T + R_{k+1}$
  $R_{k+2} \leftarrow R_{k+2} + \varepsilon$
end for
$AB[k] \leftarrow R_k$
return $AB[k]$, R_{k+1}, R_{k+2}

Listing 1.1. ARM code for calculating a column in product scanning.

```
@ k = 6
@ r5 and r4 hold R_6, R_7 respectively
@ r6, r7, r8 hold A[3], A[4] and A[5] respectively
@ r9, r10, r11 hold B[3], B[1], B[2] respectively
MOV   r12, #0
MOV   r3,  #0
UMLAL r5,  r12, r8, r10 @ A5 B1
ADDS  r4,  r4,  r12
ADC   r3,  r3,  #0
MOV   r14, #0
UMLAL r5,  r14, r7, r11 @ A4 B2
ADDS  r4,  r4,  r14
ADC   r3,  r3,  #0
MOV   r12, #0
UMLAL r5,  r12, r6, r9 @ A3 B3
ADDS  r4,  r4,  r12
ADC   r3,  r3,  #0
@ r5 holds AB[6], r4 holds R_7, @ r3 holds R_8
```

One possible optimization is **delaying the carry bit**: eliminating the last addition of Algorithm 1, this addition can be deferred to the next column with the use of a single instruction to add the partial products and the carry bit. This is easier on ARM processors, where there is fine-grained control of whether or not instructions may update the processor flags. Other optimizations involve proper register allocation in order to avoid reloads, saving up a few cycles.

Carry Elimination. Storing partial products in extra registers without adding them avoids potential carry values. 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 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 carry 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. Algorithm 2 and Listing 1.2 illustrate how a column from product-scanning can be evaluated following this strategy.

Algorithm 2. Column computation in product scanning, eliminating carries.

Input: Operands A, B; column index k; m partial products $R_k[0, \ldots, m-1]$ (calculated during column $k-1$ and stored in registers).

Output: Partial product $AB[k]$; m partial products $R_{k+1}[0, \ldots, m-1]$ (higher half part of the calculated partial product for column $k+1$ stored in registers).

$t \leftarrow 1$
for all $(i, j) \mid i + j = k, 0 \leq i < j \leq n-1$ **do**
  $(R_k[t]R_k[0]) \leftarrow A[i] \times B[j] + R_k[0] + R_k[t]$
  $R_{k+1}[t-1] \leftarrow R_k[t]$
  $t \leftarrow t + 1$
end for
$AB[k] \leftarrow R_k[0]$
return $AB[k]$, $R_{k+1}[0, \ldots, m-1]$

Listing 1.2. ARM code for calculating a column in product scanning without carries.

```
@ k = 6
@ r3, r4, r12 and r5 hold R_6[0,1,2,3]
@ r6, r7, r8 hold A[3], A[4] and A[5] respectively
@ r9, r10, r11 hold B[3], B[1], B[2] respectively
UMAAL r3, r4,  r8, r10 @ A5 B1
UMAAL r3, r12, r7, r11 @ A4 B2
UMAAL r3, r5,  r6, r9  @ A3 B3
@ r3 holds (partially) AB[6]
@ r4, r5 and r12 hold partial products for k = 7
```

Note that this strategy is limited by the number of working registers available. These registers hold partial products without adding them up, avoiding the need of carry handling, so strategies diving columns into rows like in Operand Caching are desirable.

4.2 Squaring

The literature suggests the use of a multiplication algorithm similar to Schoolbook [30], but saving up registers and repeated multiplications. Due to its similarity with product scanning (and the possibility to apply the above optimization techniques), we choose the Sliding Block Doubling algorithm as squaring routine.

Note that, with the usage of carry flag present in some machine architectures, both *Sliding Block Doubling* and the *Bottom Line* steps (as described in Sect. 3) can be efficiently computed. In order to avoid extra memory access, one can implement those two routines 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. We propose a technique to alleviate the register pressure: calculating a few multiplications akin to the Initial Block step as presented in the Operand Caching reduces register usage, allowing proper carry catching and handling in exchange for a few memory accesses (Fig. 4).

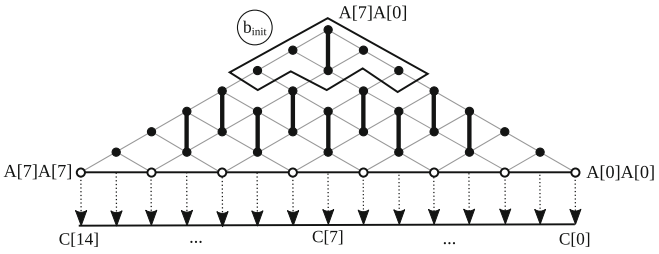

Fig. 4. Sliding Block Doubling with Initial Block

In this example, each product-scanning column is limited to height 2, meaning that only two consecutive multiplications can be handled without losing partial products. Incrementing the size of the "initial block" (or, more accurately, the initial triangle) frees up registers during the bottom row evaluation.

5 Elliptic Curves

An elliptic curve E over a field $\mathbb{F}_q$ is the set of solutions $(x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ which satisfy the Weierstrass equation

$$E/\mathbb{F}_q : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \tag{1}$$

where $a_1, a_2, a_3, a_4, a_6 \in \mathbb{F}_q$ and the curve discriminant is $\Delta \neq 0$. We restrict our attention to curves defined over prime fields which can be represented in the Montgomery [24] (or Twisted Edwards [4]) model, allowing faster formulas and unified arithmetic [6].

The set of points $E(\mathbb{F}_q) = \{(x, y) \in E(\mathbb{F}_q)\} \cup \{\mathcal{O}\} = \{P \in E(\mathbb{F}_q)\} \cup \{\mathcal{O}\}$ under the addition operation $+$ (chord and tangent) forms an additive group, with $\mathcal{O}$ as the identity element. Given an elliptic curve point $P \in E(\mathbb{F}_q)$ and an integer k, the operation kP, called scalar point multiplication, is defined by the addition of the point P to itself $k - 1$ times. This operation encodes the security assumption for Elliptic Curve Cryptography (ECC) protocols, basing their security on the hardness of solving the elliptic curve analogue of the discrete logarithm problem (ECDLP). Given a public key represented as a point Q in the curve, the problem amounts to finding the secret $k \in \mathbb{Z}$ such that $Q = kP$ for some given point P in the curve.

ECC is an efficient yet conservative option for deploying public-key cryptography in embedded systems, since the ECDLP still enjoys conjectured full-exponential security against classical computers and, consequently, reduced key sizes and storage requirements. In practice, a conservative instance of this problem can be obtained by selecting prime curves of near-prime order without supporting any non-trivial endomorphisms. Curve25519 is a popular curve at the 128-bit security level represented through the Montgomery model

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

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. Using a birational equivalence, Curve25519 can be also represented in the twisted Edwards model using full coordinates to allow instantiations of secure signature schemes:

$$\text{edwards25519:} \quad -x^2 + y^2 = 1 - \frac{121655}{121666}x^2y^2. \tag{3}$$

Key exchange protocols and digital signature schemes are building blocks for applications like key distribution schemes and secure software updates based on code signing. These protocols are fundamental for preserving the integrity of software running in embedded devices and establishing symmetric cryptographic keys for data encryption and secure communication.

5.1 Elliptic Curve Diffie Hellman

The Elliptic Curve Diffie Hellman protocol is an instantiation of the Diffie-Hellman key agreement protocol over elliptic curves. Modern implementations of this protocol employ x-coordinate-only formulas over a Montgomery model of the curve, for both computational savings, side-channel security and ease of implementation. Following this idea, the protocol may be implemented using the X25519 function, which is in essence a scalar multiplication of a point on the

Curve25519 [2]. In this scheme, a pair of entities generate their respective private keys, each of them 32-byte long. A public, generator point P is multiplied by the private key, generating a public key. Then, those entities exchange their public keys over an insecure channel; computing the X25519 function with their private keys and the received point generates a shared secret which may be used to generate a symmetric session key for both parties.

Since the ECDH protocol does not authenticate keys, public key authentication must be performed off-band, or an authenticated key agreement scheme such as the Elliptic Curve Menezes-Qu-Vanstone (ECMQV) [21] must be adopted.

For data confidentiality, authenticated encryption can be constructed by combining X25519 as an interactive key exchange mechanism, together with a block or stream cipher and a proper mode of operation, as proposed in the future Transport Layer Security protocol versions. Alternatively, authenticated encryption with additional data (AEAD) schemes may be combined with X25519, replacing block ciphers and a mode of operation.

5.2 Ed25519 Digital Signatures

The Edwards-curve Digital Signature Algorithm [5] (EdDSA) is a signature scheme variant of Schnorr signatures based on elliptic curves represented in the Edwards model. Like other discrete-log based signature schemes, EdDSA requires a secret value, or nonce, unique to each signature. For reducing the risk of a random number generator failure, EdDSA calculates this nonce deterministically, as the hash of the message and the private key. Thus, the nonce is very unlikely to be repeated for different signed messages. While this reduces the attack surface in terms of random number generation and improves nonce misuse resistance during the signing process, high quality random numbers are still needed for key generation. When instantiated using edwards25519 (Eq. 3), the EdDSA scheme is called Ed25519. Concretely, let H be the SHA512 hash function mapping arbitrary-length strings to 512-bit hash values. The signature of a message M under this scheme and private key k is the 512-bit string (R, S), where $R = rB$, for B a generator of the subgroup of points or order ℓ and r computed as $H(H(k), M)$; $S = r + H(R, A = aB, M) \bmod \ell$, for an integer a derived from $H(k)$. Verification works by parsing the signature components and checking if the equation $SB = R + H(R, A, M)A$ holds [5].

6 Implementation Details and Results

The focus given in this work is microcontrollers suitable for integration within embedded projects. Therefore, we choose some representative ARM architecture processors. Specifically, the implementations were benchmarked on the following platforms:

– **Teensy**: Teensy 3.2 board equipped with a MK20DX256VLH7 Cortex-M4-based microcontroller, clocked at 48 and 72 MHz.

- **STM32F401C**: STM32F401 Discovery board powered by a STM32F401C microcontroller, also based on the Cortex-M4 design, clocked at 84 MHz.
- **Cortex-A7/A15**: ODROID-XU4 board with a Samsung Exynos5422 CPU clocked at 2 GHz, containing four Cortex-A7 and four Cortex-A15 cores in a heterogeneous configuration.

Code for the Teensy board was generated using GCC version 5.4.1 compiled with the `-O3 -mthumb` flags; same settings apply for code compiled to the STM32F401C board, but using an updated compiler version (7.2.0). For the Cortex-A family, code was generated with GCC version 6.3.1 using the `-O3` optimization flag. Cycle counts were obtained using the corresponding cycle counter in each architecture. Randomness, where required, was sampled through `/dev/urandom` on the Cortex-A7/A15 device. In the Cortex-M4 boards, NIST's `Hash_DRBG` was implemented with SHA256 and the generator is seeded by analogically sampling disconnected pins on the board.

Albeit not the most efficient for every possible target, the codebase is the same for every ARMv7 processor equipped with DSP instructions, being ideal to large heterogeneous deployments, such as a network of smaller sensors connected to a larger central server with a more powerful processor than its smaller counterparts. This helps code maintenance, avoiding possible security problems.

6.1 Field Arithmetic

Table 1 presents timings and Table 3 presents code size for field operations with implementation described in Sect. 4. In comparison to the current state-of-art [28], 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. The multiplication routine may be used in replacement of the squaring if code size is a restriction, since 1 **S** is approximately 0.9 **M**. Implementation of all arithmetic operations take less code space in comparison to [28], ranging from 20% savings in the addition to 50% from the multiplier.

As noted by Hasse [14], cycle counts on the same Cortex-M4-based controller can be different depending on the clock frequency set on the chip. Different clock frequencies set for the controller and the memory may cause stalls on the former if the latter is slower. For example, the multiplication and the squaring implementations, which rely on memory operations, use 10% more cycles when the controller is set to a 33% higher frequency. This behavior is also present on cryptographic schemes, as shown in Table 2.

6.2 X25519 Implementation

X25519 was implemented using the standard Montgomery ladder over the x-coordinate. Standard tricks like randomized projective coordinates (amounting to a 1% performance penalty) and constant-time conditional swaps were implemented for side-channel protection. Cycle counts of the X25519 function executed on the evaluated processors are shown in Table 2 and code size in Table 3.

Table 1. Timings in cycles for arithmetic in $\mathbb{F}_{2^{255}-19}$ on multiple ARM processors. Numbers for this work were taken as the average of 256 executions.

	Cortex	Add/Sub	Mult	Mult by word	Square	Inversion
De Groot [12]	M4	73/77	631	129	563	151997
De Santis [28]	M4	106	546	**72**	362	96337
This work	M4 @ 48 MHz (Teensy)	**86**	**276**	76	**252**	**66634**
	M4 @ 72 MHz (Teensy)	**86**	**310**	76	**280**	**75099**
	M4 @ 84 MHz (STM32F401C)	**86**	**273**	76	**243**	**64425**
	A7	52	290	61	233	62648
	A15	36	225	37	139	41978
	Cortex	$\mathbb{F}_{p^2}$ **Add/Sub**	$\mathbb{F}_{p^2}$ **Mult**	**Mult by word**	$\mathbb{F}_{p^2}$ **Square**	$\mathbb{F}_{p^2}$ **Inversion**
FourQ [22]	M4 (STM32F407)	84/86	358	-	215	21056

Table 2. Timings in cycles for computing the Montgomery ladder in the X25519 key exchange; and key generation, signature and verification of a 5-byte message in the Ed25519 scheme. Key generation encompasses taking a secret key and computing its public key; signature takes both keys and a message to generate its respective signature. Numbers were taken as the average of 256 executions in multiple ARM processors. Protocols are inherently protected against timing attacks (constant-time – CT) on the Cortex-M4 due to the lack of cache memory, while side-channel protection is explicitly needed in the Cortex-A. Performance penalties for side-channel protection can be obtained by comparing the implementations with CT = Y over N in the same platform.

	CT	Cortex	X25519	Ed25519 Key Gen.	Ed25519 Sign	Ed25519 Verify
De Groot [12]	Y	M4	1816351	-	-	-
De Santis [28]	Y	M4	1563852	-	-	-
This work	Y	M4 @ 48 MHz (Teensy)	**907240**	**347225**	**496039**	**1265078**
	Y	M4 @ 72 MHz (Teensy)	**1003707**	**379734**	**531471**	**1427923**
	Y	M4 @ 84 MHz (STM32F401)	**894391**	**389480**	**543724**	**1331449**
Schwabe, Bernstein [7]	Y	A8	**527102**	-	**368212**	**650102**
This work	N	A7	-	-	423058	1118806
	Y	A7	825914	397261	524804	-
	N	A15	-	-	264252	776806
	Y	A15	572910	245377	305797	-
eBACS ref. code [10]	Y	A15	**342477**	**241641**	**245712**	**730047**
	CT	**Cortex**	**DH**	**SchnoorQ Key Gen.**	**SchnoorQ Sign**	**SchnoorQ Verify**
FourQ [22]	Y	M4 (STM32F407)	542900	265100	345400	648600

Table 3. Code size in bytes for implementing arithmetic in $\mathbb{F}_{2^{255}-19}$, X25519 and Ed25519 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 Groot [12]	44	64	1284	300	1168
De Santis [28]	138	148	1264	116	882
This work	110	108	**622**	**92**	**562**
	Inversion	**X25519**	**Ed25519 Key Gen.**	**Ed25519 Sign**	**Ed25519 Verify**
De Groot [12]	388	4140	-	-	-
De Santis [28]	484	**3786**	-	-	-
This work	**328**	4152	**21265**	**22162**	**28240**

Our implementation is 42% faster than De Santis and Sigl [28] while staying competitive in terms of code size.

Note on Conditional Swaps. The classical conditional swap using logic instructions is used by default as the compiler optimizes it using function inlining, saving about 30 cycles. However, this approach opens a breach for a power analysis attack, as shown in [25], since all bits from a 32-bit long register (in ARM architectures) must be set or not depending on a secret bit.

Alternatively, the conditional swap operation can be implemented by setting the 4-bit `ge`-flag in the Application Program Status Register (`ASPR`) and then issuing the `SEL` instruction, which pick parts from the operand registers in byte-sized blocks and writes them to the destination [1]. Note that setting `0x0` to the `ASPR.ge` flag and issuing `SEL` copies one of the operands; setting `0xF` and using `SEL` copies the other one. The `ASPR` cannot be set directly through a `MOV` with an immediate operand, so a Move to Special Register (`MSR`) instruction must be issued. Only registers may be used as arguments of this operation, so another one must be used to set the `ASPR.ge` flag. Therefore, at least 8 bits must be used to implement the conditional move. This theoretically reduces the attack surface of a potential side-channel analysis, down from 32 bits.

6.3 Ed25519 Implementation

Key generation and message signing requires a fixed-point scalar multiplication, here implemented through a comb-like algorithm proposed by Hamburg in [15]. The signed-comb approach recodes the scalar into its signed binary form using a single addition and a right-shift. This representation is divided in blocks and each one of those are divided in combs, much like in the multi-comb approach described in [16]. Like in the original work, we use five teeth for each of the five

blocks and 10 combs for each block (11 for the last one) due to the performance balance between the direct and the linear table scan to access precomputed data if protection against cache attacks is required. To effectively calculate 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 [16], with a window of width 5 for the A point, generated on-the-fly, taking approximately 3 KiB of volatile memory. 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. Note that verification has no need to be executed in constant time, since all input data is (expected to be) public. Decoding uses a standard field exponentiation for both inversion and square root to calculate the y-coordinate as suggested by [19] and [5]; this exponentiation is carried out by the Itoh-Tsujii algorithm, providing an efficient way to calculate the missing coordinate. Timings for computing a signature (both protected and unprotected against cache attacks) and verification functionality in the evaluated processors can be found in Table 2. Arithmetic modulo the group order in Ed25519-related operations relates closely to the previously shown arithmetic modulo $2^{255} - 19$, but Barrett reduction is used instead.

Final Remarks. We consider that our implementation is competitive in comparison to the mentioned works in Sect. 3, given the performance numbers shown in Tables 2 and 3. 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.

Acknowledgments. The authors gratefully acknowledge financial support from LG Electronics Inc. during the development of this work, under project "*Efficient and Secure Cryptography for IoT*", and Armando Faz-Hernández for his helpful contributions and discussions during its development. We also thank the anonymous reviewers for their helpful comments.

References

1. ARM: Cortex-M4 Devices Generic User Guide (2010). http://infocenter.arm.com/help/index.jsp?topic=%2Fcom.arm.doc.dui0553a%2FCHDBFFDB.html
2. Bernstein, D.J.: Curve25519: new diffie-hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 207–228. Springer, Heidelberg (2006). https://doi.org/10.1007/11745853_14
3. Bernstein, D.J.: 25519 naming, August 2014. https://www.ietf.org/mail-archive/web/cfrg/current/msg04996.html
4. Bernstein, D.J., Birkner, P., Joye, M., Lange, T., Peters, C.: Twisted edwards curves. In: Vaudenay, S. (ed.) AFRICACRYPT 2008. LNCS, vol. 5023, pp. 389–405. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68164-9_26
5. Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.: High-speed high-security signatures. J. Crypt. Eng. **2**(2), 77–89 (2012)

6. Bernstein, D.J., Lange, T.: Analysis and optimization of elliptic-curve single-scalar multiplication. Contemp. Math. Finite Fields Appl. **461**, 1–20 (2008)
7. Bernstein, D.J., Schwabe, P.: NEON crypto. In: Prouff, E., Schaumont, P. (eds.) CHES 2012. LNCS, vol. 7428, pp. 320–339. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33027-8_19
8. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 37–51. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-69053-0_4
9. Costello, C., Longa, P.: Four$\mathbb{Q}$: four-dimensional decompositions on a $\mathbb{Q}$-curve over the mersenne prime. In: Iwata, T., Cheon, J.H. (eds.) ASIACRYPT 2015. LNCS, vol. 9452, pp. 214–235. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48797-6_10
10. Bernstein, D.J., Lange, T. (eds.) eBACS: ECRYPT Benchmarking of Cryptographic Systems. https://bench.cr.yp.to
11. Düll, M., et al.: High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. Des. Codes Crypt. **77**(2–3), 493–514 (2015)
12. de Groot, W.: A Performance Study of X25519 on Cortex-M3 and M4. Ph.D. thesis, Eindhoven University of Technology, September 2015
13. Großschädl, J., Oswald, E., Page, D., Tunstall, M.: Side-channel analysis of cryptographic software via early-terminating multiplications. In: Lee, D., Hong, S. (eds.) ICISC 2009. LNCS, vol. 5984, pp. 176–192. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14423-3_13
14. Haase, B.: Memory bandwidth influence makes cortex m4 benchmarking difficult, September 2017. https://ches.2017.rump.cr.yp.to/fe534b32e52fcacee-026786ff44235f0.pdf
15. Hamburg, M.: Fast and compact elliptic-curve cryptography. IACR Crypt. ePrint Arch. **2012**, 309 (2012)
16. Hankerson, D., Menezes, A.J., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer-Verlag New York Inc., Secaucus (2003). https://doi.org/10.1007/b97644
17. Hutter, M., Wenger, E.: Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 459–474. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23951-9_30
18. Jao, D., De Feo, L.: Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In: Yang, B.-Y. (ed.) PQCrypto 2011. LNCS, vol. 7071, pp. 19–34. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25405-5_2
19. Josefsson, S., Liusvaara, I.: Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, January 2017. https://rfc-editor.org/rfc/rfc8032.txt
20. Kocher, P.C.: Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-68697-5_9
21. Law, L., Menezes, A., Qu, M., Solinas, J.A., Vanstone, S.A.: An efficient protocol for authenticated key agreement. Des. Codes Crypt. **28**(2), 119–134 (2003)
22. Liu, Z., Longa, P., Pereira, G.C.C.F., Reparaz, O., Seo, H.: Four$\mathbb{Q}$ on embedded devices with strong countermeasures against side-channel attacks. In: Fischer, W., Homma, N. (eds.) CHES 2017. LNCS, vol. 10529, pp. 665–686. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66787-4_32
23. Liu, Z., Seo, H., Kim, H.: A synthesis of multi-precision multiplication and squaring techniques for 8-bit sensor nodes: state-of-the-art research and future challenges. J. Comput. Sci. Technol. **31**(2), 284–299 (2016)

24. Montgomery, P.L.: Speeding the pollard and elliptic curve methods of factorization. Math. Comput. **48**(177), 243–264 (1987). https://doi.org/10.2307/2007888
25. Nascimento, E., Chmielewski, L., Oswald, D., Schwabe, P.: Attacking embedded ECC implementations through cmov side channels. IACR Crypt. ePrint Arch. **2016**, 923 (2016)
26. Oliveira, T., López, J., Hışıl, H., Faz-Hernández, A., Rodríguez-Henríquez, F.: How to (Pre-)compute a ladder. In: Adams, C., Camenisch, J. (eds.) SAC 2017. LNCS, vol. 10719, pp. 172–191. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-72565-9_9
27. Renes, J., Smith, B.: qDSA: small and secure digital signatures with curve-based diffie-hellman key pairs. IACR Crypt. ePrint Arch. **2017**, 518 (2017)
28. Santis, F.D., Sigl, G.: Towards side-channel protected X25519 on ARM cortex-M4 processors. In: SPEED-B. Utrecht, The Netherlands, October 2016. http://ccccspeed.win.tue.nl/
29. Seo, H., Kim, H.: Consecutive operand-caching method for multiprecision multiplication, revisited. J. Inform. Commun. Convergence Eng. **13**(1), 27–35 (2015)
30. Seo, H., Liu, Z., Choi, J., Kim, H.: Multi-precision squaring for public-key cryptography on embedded microprocessors. In: Paul, G., Vaudenay, S. (eds.) INDOCRYPT 2013. LNCS, vol. 8250, pp. 227–243. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03515-4_15