Des. Codes Cryptogr. (2015) 77:493–514
DOI 10.1007/s10623-015-0087-1

High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers

Michael Düll[1] · Björn Haase[2] · Gesine Hinterwälder[1] · Michael Hutter[3] · Christof Paar[1] · Ana Helena Sánchez[4] · Peter Schwabe[4]

Received: 16 November 2014 / Revised: 17 April 2015 / Accepted: 19 April 2015 /
Published online: 31 May 2015

Abstract This paper presents new speed records for 128-bit secure elliptic-curve Diffie–Hellman key-exchange software on three different popular microcontroller architectures. We

This is one of several papers published in *Designs, Codes and Cryptography* comprising the "Special Issue on Cryptography, Codes, Designs and Finite Fields: In Memory of Scott A. Vanstone".

This work was supported by the Austrian Science Fund (FWF) under the grant number TRP251-N23, by the Netherlands Organisation for Scientific Research (NWO) through Veni 2013 project 13114, by the European Cooperation in Science and Technology (COST) Action IC1204 (Trustworthy Manufacturing and Utilization of Secure Devices—TRUDEVICE), and by the German Federal Ministry for Economic Affairs and Energy (Grant 01ME12025 SecMobil). Work was done while Michael Hutter was with Graz University of Technology, Austria. Permanent ID of this document: `bd41e6b96370dea91c5858f1b809b581`.

✉ Peter Schwabe
peter@cryptojedi.org

Michael Düll
michael.duell@rub.de

Björn Haase
info@conducta.endress.com

Gesine Hinterwälder
gesine.hinterwaelder@rub.de

Michael Hutter
michael.hutter@cryptography.com

Christof Paar
christof.paar@rub.de

Ana Helena Sánchez
saralup@gmail.com

1 Horst Görtz Institute for IT-Security, Ruhr-University Bochum, 44801 Bochum, Germany

2 Endress+Hauser Conducta GmbH+Co. KG, Dieselstraße 24, 70839 Gerlingen, Germany

3 Cryptography Research, 425 Market Street, 11th Floor, San Francisco, CA 94105, USA

4 Digital Security Group, Radboud University, PO Box 9010, 6500 GL Nijmegen, The Netherlands

consider a 255-bit curve proposed by Bernstein known as Curve25519, which has also been adopted by the IETF. We optimize the X25519 key-exchange protocol proposed by Bernstein in 2006 for AVR ATmega 8-bit microcontrollers, MSP430X 16-bit microcontrollers, and for ARM Cortex-M0 32-bit microcontrollers. Our software for the AVR takes only 13,900,397 cycles for the computation of a Diffie–Hellman shared secret, and is the first to perform this computation in less than a second if clocked at 16 MHz for a security level of 128 bits. Our MSP430X software computes a shared secret in 5,301,792 cycles on MSP430X microcontrollers that have a 32-bit hardware multiplier and in 7,933,296 cycles on MSP430X microcontrollers that have a 16-bit multiplier. It thus outperforms previous constant-time ECDH software at the 128-bit security level on the MSP430X by more than a factor of 1.2 and 1.15, respectively. Our implementation on the Cortex-M0 runs in only 3,589,850 cycles and outperforms previous 128-bit secure ECDH software by a factor of 3.

Keywords Curve25519 · ECDH key-exchange · Elliptic-curve cryptography · Embedded devices · AVR ATmega · MSP430 · ARM Cortex-M0

1 Introduction

A large and growing share of the world's CPU market is formed by embedded microcontrollers. A surprisingly large number of embedded systems require security, e.g., electronic passports, smartphones, car-to-car communication and industrial control units. The continuously growing Internet of Things will only add to this development. It is of great interest to provide efficient cryptographic primitives for embedded CPUs, since virtually every security solution is based on crypto algorithms. Whereas symmetric algorithms are comparably efficient and some embedded microcontrollers even offer hardware support for them [3], asymmetric cryptography is notoriously computational intensive.

Since the invention of elliptic-curve cryptography (ECC) in 1985, independently by Koblitz [26] and Miller [31], it has become the method of choice for many applications, especially in the embedded domain. Compared to schemes that are based on the hardness of integer factoring, most prominently RSA, and schemes based on the hardness of the discrete logarithm in the multiplicative group $\mathbb{Z}_n^*$, like the classical Diffie–Hellman key exchange or DSA, ECC offers significantly shorter public keys, faster computation times for most operations, and an impressive security record. For suitably chosen elliptic curves, the best attacks known today still have the same complexity as the best attacks known in 1985. Over the last one and half decade or so, various elliptic curves have been standardized for use in cryptographic protocols such as TLS. The most widely used standard for ECC are the NIST curves proposed by NSA's Jerry Solinas and standardized in [33, Appendix D]. Various other curves have been proposed and standardized, for example the FRP256v1 curve by the French ANSSI [1], the Brainpool curves by the German BSI [30], or the SM2 curves proposed by the Chinese government [36].

It is known for quite a while that all of these standardized curves are not optimal from a performance perspective and that special cases in the group law complicate implementations that are at the same time correct, secure, and efficient. These disadvantages together with some concerns about how these curves were constructed—see, for example [10,37]—recently lead to increased interest in reconsidering the choice of elliptic curves for cryptography. As a consequence, in 2015 the IETF adopted two next-generation curves as draft internet standard for usage with TLS [25]. One of the promising next-generation elliptic curves now also adopted by the IETF is Curve25519. Curve25519 is already in use in various applications

today and was originally proposed by Bernstein in 2006 [5]. Bernstein uses the Montgomery form of this curve for efficient, secure, and easy-to-implement elliptic-curve Diffie–Hellman key exchange. Originally, the name "Curve25519" referred to this key-exchange protocol, but Bernstein recently suggested to rename the scheme to X25519 and to use the name Curve25519 for the underlying elliptic curve [6]. We will adopt this new notation in this paper.

Several works describe the excellent performance of this key-agreement scheme on large desktop and server processors, for example, the Intel Pentium M [5], the Cell Broadband Engine [13], ARM Cortex-A8 with NEON [7], or Intel Nehalem/Westmere [8,9].

1.1 Contributions of this paper

This paper presents implementation techniques of X25519 for three different, widely used embedded microcontrollers. All implementations are optimized for high speed, while executing in constant time, and they set new speed records for constant-time variable-base-point scalar multiplication at the 128-bit security level on the respective architectures.

To some extent, the results presented here are based on earlier results by some of the authors. However, this paper does not merely collect those previous results, but significantly improves performance. Specifically, the software for the AVR ATmega family of microcontrollers presented in this paper takes only 13,900,397 cycles and is thus more than a factor of 1.6 faster than the X25519 software described by Hutter and Schwabe [23]. The X25519 implementation for MSP430Xs with 32-bit multiplier presented in this paper takes only 5,301,792 cycles and is thus more than a factor of 1.2 faster, whereas the implementation for MSP430Xs with 16-bit multiplier presented in this paper takes 7,933,296 cycles and is more than a factor of 1.15 faster than the software presented by Hinterwälder et al. [21].

Furthermore, this paper is the first to present a X25519 implementation optimized for the very widely used ARM Cortex-M0 architecture. The implementation requires only 3,589,850 cycles, which is a factor of 3 faster than the scalar multiplication on the NIST P-256 curve described by Wenger et al. [45].

1.2 A note on side-channel protection

All the software presented in this paper avoids secret-data-dependent branches and secretly indexed memory access and is thus inherently protected against timing attacks. Protection against power-analysis (and EM-analysis) attacks is more complex. For example, the implementation of the elliptic-curve scalar multiplication by Wenger et al. [45] includes an initial randomization of the projective representation (and basic protection against fault-injection attacks). The authors claim that their software is "secure against (most) side-channel attacks". Under the assumption that good randomness is readily available (which is not always the case in embedded systems), projective randomization indeed protects against first-order DPA attacks and the recently proposed online-template attacks [4]. However, it does not protect against horizontal attacks [12] or higher-order DPA attacks. DPA attacks are mainly an issue if X25519 is used for static Diffie–Hellman key exchange with long-term keys; they are not an issue at all for ephemeral Diffie–Hellman without key re-use.

Adding projective randomization would be easy (assuming a reliable source of randomness) and the cost would be negligible, but we believe that serious protection against side-channel attacks requires more investigation, which is beyond the scope of this paper.

1.3 Availability of software

We placed all the software described in this paper into the public domain. The software for AVR ATmega is available at http://munacl.cryptojedi.org/curve25519-atmega.shtml; the software for TI MSP430 is available at http://munacl.cryptojedi.org/curve25519-msp430.shtml; and the software for ARM Cortex M0 is available at http://munacl.cryptojedi.org/curve25519-cortexm0.shtml.

1.4 Organization of this paper

Section 2 reviews the X25519 elliptic-curve Diffie–Hellman key exchange protocol. Section 3 describes our implementation for AVR ATmega, Sect. 4 describes our implementation for MSP430X, and Sect. 5 describes our implementation for Cortex-M0. Each of these three sections first briefly introduces the architecture, then gives details of the implementation of the two most expensive operations, namely field multiplication and squaring, and then concludes with remarks on other operations and the full X25519 implementation. Finally, Sect. 6 presents our results and compares them to previous results.

2 Review of X25519

X25519 elliptic-curve Diffie–Hellman key-exchange was introduced in 2006 by Bernstein [5]. It is based on arithmetic on the Montgomery curve Curve25519 with equation

$$E : y^2 = x^3 + 486662x^2 + x$$

defined over the field $\mathbb{F}_{2^{255}-19}$. Computation of a shared secret, given a 32-byte public key and a 32-byte secret key, proceeds as follows: The 32-byte public key is the little-endian encoding of the x-coordinate of a point P on the curve; the 32-byte secret key is the little-endian encoding of a 256-bit scalar s. The most significant bit of this scalar is set to 0, the second-most significant bit of the scalar is set to 1, and the 3 least significant bits of the scalar are set to 0. The 32-byte shared secret is the little-endian encoding of the x-coordinate of $[s]P$. Computation of a Diffie–Hellman key pair uses the same computation, except that the public key is replaced by the fixed value 9, which is the x-coordinate of the chosen base point of the elliptic curve group.

In all previous implementations of X25519, and also in our implementations, the x-coordinate of $[s]P$ is computed by using the efficient x-coordinate-only formulas for differential addition and doubling introduced by Montgomery [32]. More specifically, the computation uses a sequence of 255 so-called "ladder steps"; each ladder step performs one differential addition and one doubling. Each ladder step is followed by a conditional swap of two pairs of coordinates. The whole computation is typically called *Montgomery ladder*; a pseudo-code description of the Montgomery ladder is given in Algorithm 1. The CSWAP function in that algorithms swaps its first two arguments X_1 and X_2 if its third argument $c = 1$. This could easily be achieved through an if-statement, but all of our implementations instead use bit-logical operations for the conditional swap to eliminate a possible timing side-channel. In all our implementations we achieve this by computing a temporary value $t = (X_1 \oplus X_2) \times c$ and further executing an XOR of this result with the original values X_1 and X_2, i.e. $X_1 = X_1 \oplus t$ and $X_2 = X_2 \oplus t$.

For the ladder-step computation we use formulas that minimize the number of temporary (stack) variables without sacrificing performance. Our implementations need stack space for

Algorithm 1 The Montgomery ladder for x-coordinate-based scalar multiplication on $E : y^2 = x^3 + 486662x^2 + x$

Input: A 255-bit scalar s and the x-coordinate x_P of some point P
Output: $(X_{[s]P}, Z_{[s]P})$ fulfilling $x_{[s]P} = X_{[s]P}/Z_{[s]P}$

$X_1 \leftarrow 1$; $Z_1 \leftarrow 0$; $X_2 \leftarrow x_P$; $Z_2 \leftarrow 1$
$p \leftarrow 0$
for $i \leftarrow 254$ downto 0 **do**
  $b \leftarrow$ bit i of s
  $c \leftarrow b \oplus p$
  $p \leftarrow b$
  $(X_1, X_2) \leftarrow \text{CSWAP}(X_1, X_2, c)$
  $(Z_1, Z_2) \leftarrow \text{CSWAP}(Z_1, Z_2, c)$
  $(X_1, Z_1, X_2, Z_2) \leftarrow \text{LADDERSTEP}(x_P, X_1, Z_1, X_2, Z_2)$
end for
return (X_1, Z_1)

Algorithm 2 Single Montgomery ladder step on Curve25519

function LADDERSTEP(x_D, X_1, Z_1, X_2, Z_2)

$T_1 \leftarrow X_2 + Z_2$
$X_2 \leftarrow X_2 - Z_2$
$Z_2 \leftarrow X_1 + Z_1$
$X_1 \leftarrow X_1 - Z_1$
$T_1 \leftarrow T_1 \cdot X_1$
$X_2 \leftarrow X_2 \cdot Z_2$
$Z_2 \leftarrow Z_2 \cdot Z_2$
$X_1 \leftarrow X_1 \cdot X_1$
$T_2 \leftarrow Z_2 - X_1$
$Z_1 \leftarrow T_2 \cdot a24$
$Z_1 \leftarrow Z_1 + X_1$
$Z_1 \leftarrow T_2 \cdot Z_1$
$X_1 \leftarrow Z_2 \cdot X_1$
$Z_2 \leftarrow T_1 - X_2$
$Z_2 \leftarrow Z_2 \cdot Z_2$
$Z_2 \leftarrow Z_2 \cdot x_D$
$X_2 \leftarrow T_1 + X_2$
$X_2 \leftarrow X_2 \cdot X_2$
return (X_1, Z_1, X_2, Z_2)

end function

only two temporary field elements. Algorithm 2 presents a pseudo-code description of the ladder step with these formulas, where $a24$ denotes the constant $(486662 + 2)/4 = 121666$.

Note that each ladder step takes 5 multiplications, 4 squarings, 1 multiplication by 121666, and a few additions and subtractions in the finite field $\mathbb{F}_{2^{255}-19}$. At the end of the Montgomery ladder, the result x is obtained in projective representation, i.e., as a fraction $x = X/Z$. X25519 uses one inversion and one multiplication to obtain the affine representation. In most (probably all) previous implementations, and also in our implementations, the inversion uses a sequence of 254 squarings and 11 multiplications to raise Z to the power of $2^{255} - 21$. The total computational cost of X25519 scalar multiplication in terms of multiplications (**M**) and squarings (**S**) is thus $255 \cdot (5\,\mathbf{M} + 4\,\mathbf{S}) + 254\,\mathbf{S} + 12\,\mathbf{M} = 1287\,\mathbf{M} + 1274\,\mathbf{S}$.

3 Implementation on AVR ATmega

3.1 The AVR ATmega family of microcontrollers

The AVR ATmega is a family of 8-bit microcontrollers. The architecture features a register file with 32 8-bit registers named `R0`, ..., `R31`. Some of these registers are special: The register pair (`R26`,`R27`) is aliased as `X`, the register pair (`R28`,`R29`) is aliased as `Y`, and the register pair (`R30`,`R31`) is aliased as `Z`. These register pairs are the only ones that can be

used as address registers for load and store instructions. The register pair (R0,R1) is special because it always holds the 16-bit result of an 8×8-bit multiplication.

The instruction set is a typical 8-bit RISC instruction set. The most important arithmetic instructions for big-integer arithmetic—and thus also large-characteristic finite-field arithmetic and elliptic-curve arithmetic—are 1-cycle addition (ADD) and addition-with-carry (ADC) instructions, 1-cycle subtraction (SUB) and subtraction-with-borrow (SBC) instructions, and the 2-cycle unsigned-multiply (MUL) instruction. Furthermore, our squaring routine (see below) makes use of 1-cycle left-shift (LSL) and left-rotate (ROL) instructions. Both instructions shift their argument to the left by one bit and both instructions set the carry flag if the most-significant bit was set before the shift. The difference is that LSL sets the least-significant bit of the result to zero, whereas ROL sets it to the value of the carry flag.

The AVR instruction set offers multiple instructions for memory access. All these instructions take 2 cycles. The LD instruction loads a value from memory to an internal general-purpose register. The ST instruction stores a value from register to memory. An important feature of the AVR is the support of pre-decrement and post-increment addressing modes that are available for the X, Y, and Z registers. For the registers Y and Z there also exist a *displacement* addressing mode where data in memory can be indirectly addressed by a fixed offset. This has the advantage that only a 16-bit base address needs to be stored in registers while the addressing of operands is done by indirect displacement and without changing the base-address value. We applied addressing with indirect displacement as much as possible in our code to increase efficiency.

AVR ATmega microcontrollers come in various different memory configurations. For example, our benchmarking platform features an ATmega2560 with 256 KB of ROM and 8 KB of RAM. Other common configurations are the ATmega128 with 128 KB of ROM and 4 KB of RAM and the ATmega328 with 32 KB of ROM and 2 KB of RAM.

All cycle counts for arithmetic operations reported in this section have been obtained from a cycle-accurate simulation (using the simulator of the Atmel AVR Studio).

3.2 Multiplication

In our AVR implementation we use an unsigned radix-2^8 representation for field elements. An element f in $\mathbb{F}_{2^{255}-19}$ is thus represented as $f = \sum_{i=0}^{31} f_i 2^{8i} \hat{=} (f_0, f_1, \ldots f_{31})$ with $f_i \in \{0, \ldots, 255\}$.

For fast 256-bit-integer multiplication on the AVR we use the recently proposed highly optimized 3-level Karatsuba multiplication routine by Hutter and Schwabe [24]. More specifically, we use the branch-free variant of their software, which is slightly slower than the "branched" variant but allows easier verification of constant-time behavior. This branch-free subtractive Karatsuba routine takes 4961 cycles without function-call overhead and thus outperforms previous results presented by Hutter and Wenger [22], and by Seo and Kim [38,39] by more than 18 %.

Not only is the Karatsuba multiplier from [24] faster than all previous work, it is also smaller than previous fully unrolled speed-optimized multiplication routines. For some applications, the size of 7616 bytes might still be considered excessive so we investigated what the time-area tradeoff is for not fully unrolling and inlining Karatsuba. A multiplier that uses 3 function calls to a 128×128-bit multiplication routine instead of fully inlining those half-size multiplication takes 5064 cycles and has a size of only 3366 bytes. Note that a single 2-level 128×128-bit Karatsuba multiplication takes 1369 cycles, therefore 957 cycles are due to the higher-level Karatsuba overhead. Because of the better speed/size trade-off, we therefore decided to integrate the latter multiplication method needing 103 cycles in

addition but saves almost 56 % of code size. Section 6 reports results for X25519 for both an implementation with the faster multiplier from [24] and the smaller and slightly slower multiplier.

The details of the size-reduced Karatsuba multiplication are as follows. Basically, we split the 256 × 256-bit multiplication into three 128 × 128-bit multiplications. We follow the notation of [24] and denote the results of these three smaller multiplications with L for the low part, H for the high part, and M for the middle part. Each of these multiplications is implemented as a 2-level refined Karatsuba multiplication and is computed via a function call named `MUL128`. This function expects the operands in the registers `X` and `Y` and the address of the result in `Z`. After the low-word multiplication L, we increment the operand and result-address pointers and perform the high-word multiplication H by a second call to `MUL128`. Note that here we do not merge the refined Karatsuba addition of the upper half of L into the computation of H as described in [24] because we would need additional conditions in `MUL128` which we avoid in general. Instead, we accumulate the higher words of L right after the computation of H. This requires the additional loading of all operands and the storing of the accumulated result back to memory—but this can be done in the higher-level Karatsuba implementation which makes our code more flexible and smaller in size. Finally, we prepare the input operands for the middle-part multiplication M by a constant-time calculation of the absolute differences and a conditional negation.

3.3 Squaring

We implemented a dedicated squaring function to improve speed of X25519. For squaring, we also made use of Karatsuba's technique but only use 2 levels and make use of some simplifications that are applicable in general. For example, in squaring many cross-product terms are equal so that the computation of those terms needs to be performed only once. These terms can then be simply shifted to the left in order to get doubled. Furthermore, it becomes obvious that by calculating the absolute difference of the input for the middle-part Karatsuba squaring M is always positive. Thus also no conditional negation is required. For squaring, we hence do not need to distinguish between a "branched" and a "branch-free" variant as opposed to the multiplication proposed in [24].

Similar to multiplication, we implemented a squaring function named `SQR128`, which is then called in a higher-level 256-bit squaring implementation. The 128-bit squaring operation needs 872 cycles. Again we use two versions of squaring, one with function calls and one fully inlined version. The fully inlined version needs a total of 3324 cycles.

3.4 Putting it together

Besides 256-bit multiplication and squaring, we implemented a separate modular reduction function as well as 256-bit modular addition and subtraction. All those implementations are implemented in assembly to obtain best performance.

During scalar multiplication in X25519, we decided to reduce all elements modulo $2^{256} - 38$ and perform a "freezing" operation at the end of X25519 to finally reduce modulo $2^{255} - 19$. This has the advantage that modular reduction is simplified throughout the entire computation because the intermediate results need not be fully reduced but can be *almost* reduced which saves additional costly reduction loops. In total, modular addition and subtraction need 592 cycles. Modular reduction needs 780 cycles.

The Montgomery arithmetic on Curve25519 requires a multiplication with the curve parameter $a24 = 121666$ (see Algorithm 2 for the usage in the Montgomery-ladder step). We specialized this multiplication in a dedicated function called `fe25519_mul121666`. It makes use of the fact that the constant has 17 bits; multiplying by this constant needs only 2 multiplication instructions and several additions per input byte. The multiplication of a 256-bit integer by 121666 needs 695 cycles. All these cycle counts are for the fully speed optimized version of our software, which unrolls all loops. Our smaller software for X25519 uses (partially) rolled loops which take a few extra cycles.

4 Implementation on MSP430X

This section describes our implementation of X25519 on MSP430X microcontrollers, which is based on and improves the software presented in [21]. We implemented X25519 for MSP430X devices that feature a 16-bit hardware multiplier as well as for those that feature a 32-bit hardware multiplier. We present execution results measured on an MSP430FR5969 [41], which has an MSP430X CPU, 64 KB of non-volatile memory (FRAM), 2 kB SRAM and a 32-bit memory-mapped hardware multiplier. The result of a 16×16-bit multiplication is available in 3 cycles on both types of MSP430X devices, those that have a 32-bit hardware multiplier as well as those that have a 16-bit hardware multiplier (cf. [41,42]). Thus, our measurement results can be generalized to other microcontrollers from the MSP430X family.

All cycle counts presented in this section were obtained when executing the code on a MSP-EXP430FR5969 Launchpad development board and measuring the execution time using the debugging functionality of the IAR Embedded Workbench IDE.

4.1 The MSP430X

The MSP430X has a 16-bit RISC CPU with 27 core instructions and 24 emulated instructions. The CPU has 16 16-bit registers. Of those, only `R4` to `R15` are freely usable working registers, and `R0` to `R3` are special-purpose registers (program counter, stack pointer, status register, and constant generator). All instructions execute in one cycle, if they operate on contents that are stored in CPU registers. However, the overall execution time for an instruction depends on the instruction format and addressing mode. The CPU features 7 addressing modes. While indirect auto-increment mode leads to a shorter instruction execution time compared to indexed mode, only indexed mode can be used to store results in RAM.

We consider MSP430X microcontrollers, which feature a memory-mapped hardware multiplier that works in parallel to the CPU. Four types of multiplications, namely signed and unsigned multiply as well as signed and unsigned multiply-and-accumulate are supported. The multiplier registers have to be loaded with CPU instructions. The hardware multiplier stores the result in two (in case of 16-bit multipliers) or four (in case of 32-bit multipliers) 16-bit registers. Further a `SUMEXT` register indicates for the multiply-and-accumulate instruction, whether accumulation has produced a carry bit. However, it is not possible to accumulate carries in `SUMEXT`. The time required for the execution of a multiplication is determined by the time that it takes to load operands to and store results from the peripheral multiplier registers.

The MSP430FR5969 (the target under consideration) belongs to a new MSP430X series featuring FRAM technology for non-volatile memory. This technology has two benefits compared to flash memory. It leads to a reduced power consumption during memory writes and further increases the number of possible write operations. However, as a drawback,

while the maximum operating frequency of the MSP430FR5969 is 16 MHz, the FRAM can only be accessed at 8 MHz. Hence, wait cycles have to be introduced when operating the MSP430FR5969 at 16 MHz. For all cycle counts that we present in this section we assume a core clock frequency of 8 MHz. Increasing this frequency on the MSP430FR5969 would incur a penalty resulting from those introduced wait cycles. Note, that this is not the case for MSP430X devices that use flash technology for non-volatile memory.

4.2 Multiplication

In our MSP430X implementation we use an unsigned radix-2^{16} representation for field elements. An element f in $\mathbb{F}_{2^{255}-19}$ is thus represented as $f = \sum_{i=0}^{15} f_i 2^{16i} \hat{=} (f_0, f_1, \ldots f_{15})$ with $f_i \in \{0, \ldots, 2^{16} - 1\}$. In order to be conform with other implementations of X25519, we consider inputs and outputs to and from the scalar multiplication on Curve25519 to be 32-byte arrays. Thus conversions to and from the used representation have to be executed at the beginning and the end of the scalar multiplication. As reduction modulo $2^{255} - 19$ requires bit shifts in the chosen representation of field elements, we reduce intermediate results modulo $2^{256} - 38$ during the entire execution of the scalar multiplication and only reduce the final result modulo $2^{255} - 19$.

Hinterwälder, Moradi, Hutter, Schwabe, and Paar presented and compared implementations of various multiplication techniques on the MSP430X architecture in [21]. They considered the carry-save, operand-caching and constant-time Karatsuba multiplication, for which they used the operand-caching technique for the computation of intermediate results. Among those implementations, the Karatsuba implementation performed best. To the best of the authors knowledge, the fastest previously reported result for 256-bit multiplication on MSP430X devices was presented by Gouvêa et al. [18]. In their work the authors have used the product-scanning technique for the multi-precision multiplication. We implemented and compared the product-scanning multiplication and the constant-time Karatsuba multiplication, and this time used the product-scanning technique for the computation of intermediate results of the Karatsuba implementation. It turns out that on devices that have a 16-bit hardware multiplier, the constant-time Karatsuba multiplication performs best. On devices that have a 32-bit hardware multiplier the product-scanning technique performs better than constant-time Karatsuba, as it makes best use of the 32-bit multiply-and-accumulate unit of the memory-mapped hardware multiplier. We thus use constant-time Karatsuba in our implementation of X25519 on MSP430X microcontrollers that have a 16-bit hardware multiplier and the product-scanning technique for our X25519 implementation on MSP430Xs that have a 32-bit hardware multiplier.

In our product-scanning multiplication implementation, where $h = f \times g \mod 2^{256} - 38$ is computed, we first compute the coefficients of the double-sized array, which results from multiplying f with g and then reduce this result modulo $2^{256} - 38$. We only have 7 general-purpose registers available to store input operands during the multiplication operation. Hence, we cannot store all input operands in working registers, but we keep as many operands in them as possible. For the computation of a coefficient of the double-sized array, which results from multiplying f by g, one has to access the contents of f in incrementing and g in decrementing order, e.g. the coefficient h_2 is computed as $h_2 = f_0 g_2 + f_1 g_1 + f_2 g_0$. As there is no indirect auto-decrement addressing mode available on the MSP430X microcontroller, we put the contents of g on the stack in reverse order at the beginning of the multiplication, which allows us to access g using indirect auto-increment addressing mode for the remaining part of the multiplication. Including function-call and reduction overhead, our 32-bit product-

scanning multiplication implementation executes in 2079 cycles on the MSP430FR5969. Without function call and modular reduction, it executes in 1693 cycles.

For MSP430X microcontrollers that have a 16-bit hardware multiplier we implemented the constant-time one-level Karatsuba multiplication (refer to Sect. 3). We use the product-scanning technique to compute the three intermediate results L, H and M. For the computation of L, H and M we have seven working registers available to store input operands. Hence, we can store almost the full input that is accessed in decrementing order in working registers and access the eighth required operand of it using indirect addressing mode. Again we first compute the double-sized array resulting from the multiplication of f and h and then reduce this result modulo $2^{256} - 38$. Our modular multiplication implementation dedicated for devices that have a 16-bit hardware multiplier executes in 3193 cycles including function call and modular reduction, and in 2718 cycles excluding those.

4.3 Squaring

In order to compute $h = f^2 \mod 2^{256} - 38$, we first compute a double-sized array resulting from squaring f and then reduce this result modulo $2^{256} - 38$. Similar to our multiplication implementation, we use the product-scanning technique for our implementation targeting devices that have a 32-bit hardware multiplier. We again store the input f on the stack in reverse order, allowing us to use indirect auto-increment addressing mode to access elements of f in decrementing order. As mentioned in Sect. 3, many multiplications of cross-product terms occur twice during the execution of the squaring operation. These do not have to be computed multiple times, but can be accounted for by multiplying an intermediate result by two, i.e. shifting it to the left by one bit. As shift operations on the result registers of the memory-mapped hardware multiplier are expensive, we move results of a multiplication back to CPU registers before executing this shift operation. Including function call and modular reduction overhead our squaring implementation executes in 1563 cycles on MSP430X microcontrollers that have a 32-bit hardware multiplier. Without reduction and function call this number decreases to 1171 cycles.

Our squaring implementation for MSP430X microcontrollers that have a 16-bit hardware multiplier follows the constant-time Karatsuba approach, where intermediate results are computed using the product-scanning technique. This function executes in 2426 cycles including function call and reduction overhead and in 1935 cycles without.

4.4 Putting it together

We implemented all finite-field arithmetic in assembly language and all curve arithmetic as well as the conversion to and from the internal representation in C.

The x-coordinate-only doubling formula requires a multiplication with the constant 121666. One peculiarity of the MSP430 hardware multiplier greatly improves the performance of the computation of $h = f \cdot 121666 \mod 2^{256} - 38$, which is that contents of the hardware multiplier's `MAC` registers do not have to be loaded again, in case the processed operands do not change. In case of having a 32-bit hardware multiplier we proceed as follows: The number 121666 can be written as $1 \cdot 2^{16} + 56130$. We store the value 1 in `MAC32H` and 56130 in `MAC32L` and then during each iteration load two consecutive coefficients of the input array f, i.e. f_i and f_{i+1} to `OP2L` and `OP2H` for the computation of two coefficients of the resulting array namely h_i and h_{i+1}. The array that results from computing f^2 is only two elements longer than the input array, which we reduce as the next step. Using this method, the

multiplication with 121666 executes in 352 cycles on MSP430s that have a 32-bit hardware multiplier, including function call and reduction.

For the 16-bit hardware multiplier version, we follow a slightly different approach. As we cannot store the full number 121666 in the input register of the hardware multiplier, we proceed as follows: To compute $h = f \cdot 121666 \mod 2^{256} - 38$ we store the value 56130 in the hardware-multiplier register `MAC`. We then compute each h_i as $h_i = f_i \cdot 56130 + f_{i-1}$ for $i \in [1 \ldots 15]$ such that we add the $(i-1)$th input coefficient to the multiplier's result registers `RESLO` and `RESHI`. This step takes care of the multiplication with $1 \cdot 2^{16}$ for the $(i-1)$th input coefficient. We further load the ith input coefficient to the register `OP2`, thus executing the multiply-and-accumulate instruction to compute the ith coefficient of the result. Special care has to be taken with the coefficient h_0, where $h_0 = f_0 \cdot 56130 + 38 \cdot f_{15}$. The method executes in 512 cycles including function call and reduction overhead.

The reduction of a double-sized array modulo $2^{256} - 38$ is implemented in a similar fashion. We store the value 38 in the `MAC`-register of the hardware multiplier. We then add the ith coefficient of the double-sized input to the result registers of the hardware multiplier and load the $(i+16)$th coefficient to the `OP2`-register. In the 32-bit version of this reduction implementation the only difference is that two consecutive coefficients can be processed in each iteration, i.e. the ith and $(i+1)$th coefficients are added to the result registers and and the $(i+16)$th and $(i+17)$th coefficient are loaded to the `OP2`-registers.

The modular addition $h = f + g \mod 2^{256} - 38$, which executes in 186 cycles on the MSP430, first adds the two most significant words of f and g. It then extracts the carry and the most significant bit of this result and multiplies those with 19. This is added to the least significant word of f. All other coefficients of f and g are added with carry to each other. The carry resulting from the addition of the second most significant words of f and g is added to the sum that was computed first.

For the computation of $h = f - g$, we first subtract g with borrow from f. If the result of the subtraction of the most significant words produces a negative result, the carry flag is cleared, while, if it produces a positive result the carry flag is set. We add this carry flag to a register `tmp` that was set to `0xffff` before, resulting in the contents of `tmp` to be `0xffff` in case of a negative result and 0 in case of a positive result of the subtraction. We AND `tmp` with 38, subtract this from the lowest resulting coefficient and ripple the borrow through. Again a possible resulting negative result of this procedure is reduced using the same method, minus the rippling of the borrow. This modular subtraction executes in the same time as the modular addition, i.e. in 199 cycles including function-call overhead.

5 Implementation on ARM Cortex-M0

5.1 The ARM Cortex M0

The ARM Cortex M0 and Cortex M0+ cores (M0) are the smallest members of ARM's recent Cortex-M series, targeting low-cost and low-power embedded devices. The M0 implements a load-store architecture. The register file consists of 16 registers `r0`, ..., `r15`, including 3 special-purpose registers for the program counter (pc) in `r15`, the return addresses (lr) in `r14`, and the stack pointer (sp) in `r13`.

Unlike its larger brothers from the ARM Cortex M series, the M0 encodes arithmetic and logic instructions exclusively in 16 bits. This 16-bit instruction encoding results in constraints with respect to register addressing. As a result, the eight lower registers `r0`, ..., `r7` can be

used much more flexibly than the upper registers `r8`, ..., `r14`. More specifically, only the lower registers `r0`, ..., `r7` may be used for pointer-based memory accesses, as destination of a load or source of a store, and for holding memory-address information. Also almost all arithmetic and logic instructions like addition and subtraction only accept lower registers as operands and results. The upper registers are mainly useful as fast temporary storage, i.e., in register-to-register-move instructions.

The M0 core supports a multiplication instruction which receives two 32-bit operands and produces a 32-bit result. Note that this is substantially different from the AVR ATmega and the MSP430X; on the M0 the upper half of the 64-bit result is cut off. For our purpose of fast multi-precision integer arithmetic, we consider the multiplier as a 16-bit multiplier. The main difference to AVR and MSP430X is then, that the result is produced in only one register. The M0 is available in two configurations, where multiplication either costs 1 cycle or 32 cycles. In this paper we focus on M0 systems featuring the single-cycle hardware multiplier, a design choice present on most M0 implementations that we are aware of. All arithmetic and logic operations, including the multiplication operate on 32-bit inputs and outputs. They all require a single clock cycle.

The M0 uses a von Neumann memory architecture with a single bus being used for both, code and data. Consequently all load and store instructions require one additional cycle for the instruction fetch. This constitutes one of the key bottlenecks to consider for the implementation of the arithmetic algorithms. Since a typical load/store instruction requires 2 cycles, while an arithmetic or multiplication operation only takes a single cycle, it is very important to make best usage of the limited memory bandwidth. Consequently it is part of our strategy to make loads and stores always operate on full 32-bit operands and use the load and store multiple (LDM/STM) instructions wherever possible. These LDM/STM instructions transfer n (up to eight) 32-bit words in one instruction, with a cost of only $n + 1$ cycles.

Like the other two platforms considered in this paper, the ARM Cortex-M0 also comes in very different memory configurations. The STM32F0-Value chips have between 16 and 256 KB of ROM and between 4 and 32 KB of RAM. For our benchmarks and tests we used a development board with an STM32F051R8T6 microcontroller with 64 KB of ROM and 8 KB of RAM. All cycle counts for arithmetic operations reported in this section have been obtained using the systick counter on this development board.

In comparison to the other architectures discussed in this paper, the M0 platform benefits from its single-cycle $32 \times 32 \rightarrow 32$-bit multiplication instruction that directly operates on the general-purpose register file. The weakness of this architecture is its slow memory interface and the restrictions resulting from the 16-bit encoding of instructions: the small register set of only 8 registers `r0`, ..., `r7` that can be used in arithmetic instructions and memory access.

5.2 Multiplication

In our Cortex-M0 implementation we use an unsigned radix-2^{32} representation for field elements. An element f in $\mathbb{F}_{2^{255}-19}$ is thus represented as $f = \sum_{i=0}^{7} f_i 2^{32i} \hat{=} (f_0, f_1, \ldots f_7)$ with $f_i \in \{0, \ldots, 2^{32} - 1\}$.

It turns out that the most efficient strategy for multiplication of $n = 256$-bit operands is a three-level refined Karatsuba method. To obtain a constant-time behavior and avoid the carry propagation, we use a variant of subtractive Karatsuba. The n-bit input operands $A = A_\ell + 2^{n/2} A_h$ and $B = B_\ell + 2^{n/2} B_h$ are first decomposed into a lower and a higher half. Then one computes the partial products $L = A_h \cdot B_h$ and and $H = A_h \cdot B_h$. The subtractive Karatsuba formulas involve a product term $M = (A_\ell - A_h) \cdot (B_\ell - B_h)$ which may be either positive or negative. The full result may then be calculated by use of the subtractive Karatsuba

formula $A \cdot B = L + 2^{n/2}(L + H - M) + 2^n \cdot H$. By use of the refined Karatsuba method, we reduce the storage needed to calculate the middle part M and at the same time we save several additions on each Karatsuba level. Analysis of the low-level constraints of the CPU architecture revealed that it is considerably more efficient not to use a signed multiplication yielding M directly but to first calculate the absolute value $|M| = |A_\ell - A_h| \cdot |B_\ell - B_h|$ and separately keep track of the sign t of the result. This stems mainly from the observation that sign changes (i.e. two's complements) of operands may be calculated in-place without requiring temporary spill registers.

Actually the variant in our M0 implementation swaps the difference of one factor of $|M|$, i.e., $|M| = |A_\ell - A_h| \cdot |B_h - B_\ell|$ and compensates for this by toggling the sign bit t. This makes branch-free combination of the partial results slightly more efficient. The calculation, thus, involves calculating the absolute value of the differences $|A_\ell - A_h|$ and $|B_h - B_\ell|$, the sign t and a conditional negation of the positive result $|M|$. As in the AVR implementation, we do not use any conditional branches, but instead use conditional computation of the two's complements. Note that the conditional calculation of the two's complement involves first a bitwise exclusive or operation with either 0 or -1, depending on the sign. Subsequently a subtraction operation of either -1 or 0 follows, being equivalent to addition of 1 or 0.

For our implementation, we represent the field elements as arrays of eight 32-bit words. Since the architecture only provides a precision of 16-bit on its multiplier, we obtain a 32-bit multiplication with 17 arithmetic instructions: 4 to convert the registers from 32 to 16 bits, 4 multiplications, 1 to save an extra input (multiplication overwrites one of the inputs), and 8 instructions (4 additions and 4 shifts) to add the middle part into the final result. Since the 32-bit multiplication requires at least 5 registers, register-to-register moves between the low and high part of the register file are required to perform more than one multiplication.

We obtain the 256-bit product using three 128-bit multiplications, each one with a cost of 332 cycles. The 128-bit multiplier uses three 64-bit multiplications which only take 81 cycles each. The full 256-bit multiplication requires 1294 cycles, about 700 cycles faster than a fully unrolled product-scanning multiplication.

5.3 Squaring

For squaring we also use three levels of refined subtractive Karatsuba. We use the same two observations as for the AVR to improve squaring performance compared to multiplication performance. First all of the partial results M, L and H entering the Karatsuba formula are solely determined by squaring operations, i.e. no full multiplication is involved. Conventional squaring of an operand $A = A_\ell + 2^k A_h$ would have required two squarings of the lower and upper halves A_ℓ^2 and A_h^2 and one multiplication for the mixed term $A_\ell \cdot A_h$. Aside from arithmetic simplification, a big benefit of avoiding this mixed-term multiplication is that one input operand fetch and register spills to memory may be spared because for squarings we have only one input operand. This benefit clearly outweighs the extra complexity linked to the additional additions and subtractions within the Karatsuba formula. Second it is easily observed that the sign of the operand M is known to be positive from the very beginning. The conditional sign change of the intermediate operand M is thus not necessary.

The 64-bit squaring takes 53 cycles using only seven registers; our 128-bit squaring takes only 206 cycles, with the advantage that we handle all temporary storage with the upper half of the register file, i.e. no use of the stack is required. Our 256-bit squaring algorithm requires 857 cycles for 256-bit operands, in comparison to 1110 cycles for an unrolled product-scanning squaring. As expected, the benefit of using Karatsuba is much smaller than for multiplication.

Table 1 Cycle count on ARM Cortex M0 with single-cycle multiplier for assembly optimized implementation and optimization for speed
Modular addition and multiplication with 121666 include reduction modulo $2^{256} - 38$

Operation	Clock cycles
Modular addition	109
Modular multiplication by 121666	184
Reduction modulo $2^{256} - 38$	175
256×256-Bit multiplication	1294
256-Bit squaring	857

Still the difference between squaring and multiplication is significant, clearly justifying to use a specialized squaring algorithm when optimizing for speed.

5.4 Putting it together

For multiplication and squaring we did not merge multiplication and reduction due to the high register pressure. Merging the operations would have led to many register spills. For these operations, we first implement a standard long-integer arithmetic and reduce the result in a second step. We use separate functions for multiplication and reduction

Throughout the X25519 calculation we reduce modulo $2^{256} - 38$ and even allow temporary results to reach up to $2^{256} - 1$. Full reduction is used only for the final result.

For addition, subtraction and multiplication with the curve constant 121666, we use a different strategy and reduce the result on the fly in registers before writing results back to memory. For these simple operations, it is possible to perform all of the arithmetic and reduction without requiring register spills to the stack. The cycle counts for these operations are summarized in Table 1. Multiplication with the curve constant is implemented by a combination of addition and multiplication. Since the constant has 17 significant bits, multiplication is implemented by a combination of a 16-bit multiplication and a 16-bit shift-and-add operation.

The strategy for reducing on the fly consists of two steps. First, the arithmetic operation (addition, subtraction, multiplication by 121666) is implemented on the most significant word. This generates carries in bits 255 and higher that need to be reduced. We strip off these carries resulting from the most significant word (setting bits 255 and higher of the result to zero) and merge the arithmetic for the lower words with reduction. This may result in an additional carry into the most significant word. However, these carries may readily be stored in bit 255 of the most significant word. This way a second carry chain is avoided.

6 Results and comparison

This section describes our implementation results for the X25519 Diffie–Hellman key-exchange on the aforementioned platforms. We present performance results in terms of the required clock cycles for one scalar multiplication. We furthermore report the required storage and RAM space. A full Diffie–Hellman key exchange requires one scalar multiplication of a fixed-basepoint and one variable-point scalar multiplication. Our software does not specialize fixed-basepoint scalar multiplication; the cost for a complete key exchange can thus be obtained by multiplying our cycle counts for one scalar multiplication by two. We compare our results to previous implementations of elliptic-curve scalar multiplication at the 128-bit security level (and selected high-performance implementations at slightly lower security levels) on the considered platforms.

6.1 Results and comparison on AVR ATmega

Our results for X25519 scalar multiplication on the AVR ATmega family of microcontrollers and a comparison with previous work are summarized in Table 2. As described in Sect. 3, all low-level functions are written in assembly. The high-level functionality is written in C; for compilation we used gcc-4.8.1 with compiler options `-mmcu=atmega2560 -O3 -mcall-prologues`. Unlike the cycle counts for subroutines reported in Sect. 3, all cycle counts reported for full elliptic-curve scalar multiplication reported here were measured using the built-in cycle counters on an Arduino MEGA development board with an ATmega2560 microcontroller. To achieve sufficient precision for the cycle counts, we combined an 8-bit and a 16-bit cycle counter to a 24-bit cycle counter.

Many implementations of elliptic-curve cryptography exist for the AVR ATmega; however, most of them aim at lower security levels of 80 or 96 bits. For example the TinyECC library by Liu and Ning implements ECDSA, ECDH, and ECIES on the 128-bit, 160-bit, and 192-bit SECG curves [27]. NanoECC by Szczechowiak, Oliveira, Scott, Collier, and Dahab uses the NIST K-163 curve [40]. Also recent ECC software for the AVR ATmega uses relatively low-security curves. For example, Liu et al. [28] report new speed records for elliptic-curve cryptography on the NIST P-192 curve. Also Dalin, Großschädl, Liu, Mßller, and Zhang focus on the 80-bit and 96-bit security levels for their optimized implementation of ECC with twisted Edwards curves presented in [14].

Table 2 summarizes the results for elliptic-curve variable-basepoint scalar multiplication on curves that offer at least 112 bits of security. Not only are both of our implementations more than 1.5 times faster than all previous implementations of ECC at the 128-bit security level, the small implementation is also considerably smaller than all previous implementations. As also stated in the footnote, the size comparison with the MoTE-ECC software presented by Liu, Wenger, and Großschädl [29] is not fair, because their software optimizes also fixed-basepoint scalar multiplication and claims a performance of 30,510,000 cycles for ephemeral Diffie–Hellman (one fixed-point and one variable-point scalar multiplication). Even under the assumption that this is the right measure for ECDH performance—which means that ephemeral keys are not re-used for several sessions, for a discussion, see [11, Appendix D]—our small implementations offers better speed and size than the one presented in [29]. The only implementation that is smaller than ours and offers reasonably close performance is the one by Gura, Patel, Wander, Eberle, and Chang Shantz presented in [20]; however, that implementation is using a curve that offers only 112 bits of security. The only implementation that is faster than ours is the DH software on the NIST-K233 curve by Aranha, Dahab, López, and Oliveira presented in [2]; however, this software also offers only 112 bits of security, has very large ROM and RAM consumptions, and uses a binary elliptic-curve with efficiently computable endomorphisms, which is commonly considered a less conservative choice. As pointed out in the footnote, the size comparision to [2] is also not entirely fair because their software also contains a specialized fixed-basedpoint scalar multiplication.

6.2 Results and comparison on MSP430X

Our results for Curve25519 on the MSP430X microcontroller and a comparison with related previous work are summarized in Table 3. As for the AVR comparison, we only list results that target reasonably high security levels. For our implementation we report cycle counts of the MSP430FR5969 for 8 MHz and 16 MHz. One might think that the cycle counts are independent of the frequency; however, due to the limited access frequency of the non-volatile

Table 2 Cycle counts, sizes, and stack usage of elliptic-curve scalar-multiplication software for AVR ATmega microcontrollers

Implementation	Curve	Clock cycles	Size	RAM usage
Aranha et al. [2]	NIST K-233	≈5,382,144	≈38,600 bytes[b]	≈3700 bytes
Aranha et al. [2]	NIST B-233	≈13,934,592	≈34,600 bytes[b]	≈2,200 bytes
Gura et al. [20]	NIST P-224	≈17,520,000	4812 bytes	422 bytes
Liu et al. [29]	256-bit Montgomery	≈21,078,200	14,700 bytes[b]	556 bytes
Wenger et al. [45]	NIST P-256	≈34,930,000	16,112 bytes	590 bytes
Hutter and Schwabe [23]	Curve25519	22,791,579	n/a[a]	677 bytes
This paper	Curve25519	14,146,844	9,912 bytes	510 bytes
This paper	Curve25519	13,900,397	17,710 bytes	494 bytes

[a] Size is reported only for the complete NaCl library core, not for stand-alone Curve25519
[b] Implementation also includes faster fixed-basepoint scalar multiplication

Table 3 Cycle counts, sizes, and stack usage of elliptic-curve scalar-multiplication software for MSP430X microcontrollers

Implementation	CPU	Curve	Clock cycles @8 MHz	Clock cycles @16 MHz	Size	Stack usage
With 16-bit hardware multiplier						
Wenger and Werner [44]	MSP430	NIST P-256	23,937,000	n/a	n/a	n/a
Wenger et al. [45]	MSP430	NIST P-256	22,170,000	n/a	8,378 bytes	418 bytes
Gouvêa et al. [16,18]	MSP430X	NIST P-256	7,284,377[a]	n/a	n/a	n/a
Hinterwälder et al. [21]	MSP430X	Curve25519	9,139,739	10,404,042	11,778 bytes	513 bytes
This paper	MSP430X	Curve25519	7,933,296	9,119,840	13,112 bytes	384 bytes
With 32-bit hardware multiplier						
Gouvêa et al. [16,18]	MSP430X	NIST P-256	5,321,776[a]	n/a	n/a	n/a
Hinterwälder et al. [21]	MSP430X	Curve25519	6,513,011	7,391,506	8956 bytes	495 bytes
This paper	MSP430X	Curve25519	5,301,792	5,941,784	10,088 bytes	382 bytes

[a] Note that the authors use the $4w$-NAF method for the scalar multiplication, which does not execute in constant time. In this paper we focus on a constant-time implementation to thwart timing attacks. Further the authors obtained some of the cycle counts using the IAR Embedded Workbench simulator. It turns out that this simulator does not report correct timings, if the memory-mapped hardware multiplier of the MSP430 is used

(FRAM) memory of the MSP430FR5969 (see Sect. 4), core clock frequencies beyond 8 MHz introduce wait cycles for memory access.

As mentioned in Sect. 4, all arithmetic operations in $\mathbb{F}_{2^{255}-19}$ (aside from inversion) are implemented in assembly. The high-level functionality is written in C; for compilation we used gcc-4.6.3 with compiler options `-mmcu=msp430fr5969 -O3`. All cycle counts reported in this section were obtained by measuring the cycle count when executing the code on an MSP-EXP430FR5969 Launchpad Development Kit [43], using the cycle counters of the chip, unlike Sect. 4 where cycle counts on the board were obtained using the debugging functionality of the IAR Embedded Workbench IDE. These cycle counters have a resolution of only 16-bits, which is not enough to benchmark our software. We use a divisor of 8 (i.e., the counter is increased every 8 cycles) and increase a global 64-bit variable every time an overflow interrupt of the on-chip counter is triggered. This gives us a counter with reasonable resolution and relatively low interrupt-handling overhead and makes it possible to independently reproduce our results without the use of the proprietary IAR Workbench IDE.

Naturally the implementation that makes use of the 32-bit hardware multiplier executes in fewer cycles and requires less program storage space than the implementation that only requires a 16-bit hardware multiplier. This is because fewer load and store instructions to the peripheral registers of the hardware multiplier have to be executed.

A plethora of literature describes implementations of elliptic curve cryptography on the MSP430 microcontroller architecture, while only few of those works describe an implementation at the 256-bit security level. The first implementation of ECC on an MSP430 microcontroller was presented in 2001 by Guajardo, Blümel, Krieger, and Paar. Their implementation at the 64-bit security level executes in 3.4 million clock cycles [19]. In 2009, Gouvêa and López reported speed records for 160 and 256-bit finite-field multiplications on the MSP430 needing 1586 and 3597 cycles, respectively [17]. Their 256-bit Montgomery-ladder scalar multiplication requires 20.4 million clock cycles; their 4-NAF and 5-NAF versions require 13.4 and 13.2 million cycles, respectively. In 2011, Wenger and Werner compared ECC scalar multiplications on various 16-bit microcontrollers [44]. Their Montgomery-ladder-based scalar multiplication on the NIST P-256 elliptic curve executes in 23.9 million cycles on the MSP430. Pendl, Pelnar, and Hutter presented the first ECC implementation running on the WISP UHF RFID tag the same year [35]. Their implementation of the NIST P-192 curve achieves an execution time of around 10 million clock cycles. They also reported the first 192-bit multi-precision multiplication results needing 2581 cycles. Gouvêa, Oliveira, and López reported new speed records for different MSP430X architectures in 2012 [18], improving their results from [17]. For the MSP430X architecture (with a 16-bit multiplier) their 160-bit and 256-bit finite-field multiplication implementations execute in 1299 and 2981 cycles, respectively. In 2013, Wenger, Unterluggauer, and Werner [45] presented an MSP430 clone with instruction-set extension to accelerate big-integer arithmetic. For a NIST P-256 elliptic curve, their Montgomery ladder implementation using randomized projective coordinates and multiple point validation checks requires 9 million clock cycles. Without instruction-set extensions their implementation needs 22.2 million cycles.

6.3 Results and comparison on ARM Cortex M0

Our results for Curve25519 on ARM Cortex-M0 and a comparison with related work are summarized in Table 4. As described in Sect. 5, all low-level functions for arithmetic in $\mathbb{F}_{2^{255}-19}$ (except for inversion, addition and subtraction) are implemented in assembly. It turned out that the addition and subtraction code generated by the compiler was almost

Table 4 Cycle counts, sizes, and stack usage of elliptic-curve scalar-multiplication software for ARM Cortex-M0 microcontrollers

Implementation	Curve	Clock cycles	Size	RAM usage
De Clercq et al. [15]	NIST K-233	2,762,000	n/a	n/a bytes
Wenger et al. [45]	NIST P-256	≈10,730,000	7168 Bytes	540 bytes
This paper	Curve25519	3,589,850	7,900 bytes	548 Bytes

as efficient as hand-optimized assembly. Higher-level functions are implemented in C; for compilation we used clang 3.5.0.

For C files we use a 3-stage compilation process. First we translate with `clang -fshort-enums -mcpu=cortex-m0 -mthumb -emit-llvm -c -nostdlib -ffreestanding -target arm-none-eabi -mfloat-abi=soft scalarmult.c` to obtain a `.bc` file, which is then optimized with `opt -Os -misched=ilpmin -misched-regpressure -enable-misched -inline` and further translated to a `.s` file with `llc -misched=ilpmin -enable-misched -misched-regpressure`. As a result of these settings, addition and subtraction functions were fully inlined. This improves speed in comparison to calls to assembly functions by avoiding the function call overhead (at the expense of roughly 1 KB larger code).

We obtained cycle counts from the systick cycle counter of an STM32F0Discovery development board. We also experimented with an LPC1114 Cortex-M0 chip but were unable to achieve the full performance of the Cortex-M0 even for very simple code (like a sequence of 1000 NOPs). For the "default" power profile the cycle counts we obtained were exactly a factor of 1.25 higher than expected. When switching to the "performance" profile (see [34, Sect. 7.16.5]), we achieved better performance, but still not the expected cycle counts.

ARM's Cortex-M microcontrollers are rapidly becoming the device of choice for applications that previously used less powerful 8-bit or 16-bit microcontrollers. It is surprising to see that there is relatively little previous work on speeding up ECC on Cortex-M microcontrollers and in particular on the Cortex-M0. Probably the most impressive previous work has recently been presented by De Clerq, Uhsadel, Van Herrewege, and Verbauwhede who achieve a performance of 2,762,000 cycles for variable base-point scalar multiplication on the 233-bit Koblitz curve sect233k1 [15]. This result is hard to directly compare to our result for three reasons. First the curve is somewhat smaller and targets the 112-bit security level rather than then 128-bit security level targeted by our implementation. Second the implementation in [15] is not protected against timing attacks. Third the software presented in [15] performs arithmetic on an elliptic-curve over a binary field. All the underlying field arithmetic is thus very different.

The only scientific paper that we are aware of that optimizes arithmetic on an elliptic curve over a large-characteristic prime field for the Cortex-M0 is the 2013 paper by Wenger, Unterluggauer, and Werner [45]. Their scalar multiplication on the secp256r1 curve is reported to take 10,730,000 cycles, almost exactly 3 times slower than our result.

Acknowledgments The authors would like to thank Daniel Bernstein for his suggestion to reverse an input to the modular multiplication implementation for the MSP430.

References

1. Agence nationale de la sécurité des systèmes d'information. Avis relatif aux paramètres de courbes elliptiques définis par l'Etat français. Journal officiel de la République Française, 0241, 17533 (2011). http://www.legifrance.gouv.fr/affichTexte.do?cidTexte=JORFTEXT000024668816.
2. Aranha D.F., Dahab R., López J., Oliveira L.B.: Efficient implementation of elliptic curve cryptography in wireless sensors. Adv. Math. Commun. **4**(2), 169–187 (2010).
3. Atmel Corporation: AVR1519: XMEGA-A1 Xplained Training—XMEGA Crypto Engines. 8-bit Atmel Microcontrollers Application Note (2011). http://www.atmel.com/Images/doc8405.pdf.
4. Batina L., Chmielewski, Ł., Papachristodoulou L., Schwabe P., Tunstall M.: Online template attacks. In: Meier W., Mukhopadhyay D. (eds.) Progress in Cryptology—INDOCRYPT 2014. Lecture Notes in Computer Science, vol. 21–36, p. 8885. Springer, Berlin (2014). http://cryptojedi.org/papers/#ota.
5. Bernstein D.J.: Curve25519: new Diffie-Hellman speed records. In: Yung M., Dodis Y., Kiayias A., Malkin T. (eds.) Public Key Cryptography—PKC 2006. Lecture Notes in Computer Science, vol. 3958, pp. 207–228. Springer, Berlin (2006). http://cr.yp.to/papers.html#curve25519.
6. Bernstein D.J.: 25519 naming. Posting to the CFRG mailing list (2014). https://www.ietf.org/mail-archive/web/cfrg/current/msg04996.html.
7. Bernstein D.J., Schwabe, P.: NEON crypto. In: Prouff E., Schaumont P. (eds.) Cryptographic Hardware and Embedded Systems—CHES 2012. Lecture Notes in Computer Science, vol. 7428, pp. 320–339. Springer, Berlin (2012). http://cryptojedi.org/papers/#neoncrypto.
8. Bernstein D.J., Duif N., Lange T., Schwabe P., Yang B.-Y.: High-speed high-security signatures. In: Preneel B., Takagi T (eds.) Cryptographic Hardware and Embedded Systems—CHES 2011. Lecture Notes in Computer Science, vol. 6917, pp. 124–142. Springer, Berlin (2011). see also full version [9].
9. Bernstein D.J., Duif N., Lange T., Schwabe P., Yang B.-Y.: High-speed high-security signatures. J. Cryptogr. Eng. **2**(2), 77–89 (2012). http://cryptojedi.org/papers/#ed25519, see also short version [8].
10. Bernstein D.J., Chou T., Chuengsatiansup C., Hülsing A., Lange T., Niederhagen R., van Vredendaal C.: How to manipulate curve standards: a white paper for the black hat. Cryptology ePrint Archive, Report 2014/571 (2014). http://eprint.iacr.org/2014/571, see also http://safecurves.cr.yp.to/bada55.html.
11. Bernstein D.J., Chuengsatiansup C., Lange T., Schwabe P.: Kummer strikes back: new DH speed records. In: Iwata T., Sarkar P (eds.) Advances in Cryptology—ASIACRYPT 2014. Lecture Notes in Computer Science, vol. 8873, pp. 317–337. Springer, Berlin (2014). Full version: http://cryptojedi.org/papers/#kummer.
12. Clavier C., Feix B., Gagnerot G., Roussellet M., Verneuil V.: Horizontal correlation analysis on exponentiation. In: Soriano M., Qing S., López J. (eds.) Information and Communications Security. Lecture Notes in Computer Science, vol. 6476, pp. 46–61. Springer, Berlin (2010). http://eprint.iacr.org/2003/237.
13. Costigan N., Schwabe P.: Fast elliptic-curve cryptography on the Cell Broadband Engine. In: Preneel B (ed.) Progress in Cryptology—AFRICACRYPT 2009. Lecture Notes in Computer Science, vol. 5580, pp. 368–385. Springer, Berlin (2009). http://cryptojedi.org/papers/#celldh.
14. Dalin D., Großschädl J., Liu Z., Müller V., Zhang W.: Twisted edwards-form elliptic curve cryptography for 8-bit AVR-based sensor nodes. In: Xu S., Zhao Y. (eds.) Proceeding of the 1st ACM Workshop on Asia Public-key Cryptography—AsiaPKC 2013, pp. 39–44. ACM, New York (2013). http://orbilu.uni.lu/handle/10993/14765.
15. De Clercq R., Uhsadel L., Van Herrewege A., Verbauwhede I.: Ultra low-power implementation of ECC on the ARM Cortex-M0+. In: DAC '14 Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference, pp. 1–6. ACM, New York (2014). https://www.cosic.esat.kuleuven.be/publications/article-2401.pdf.
16. Gouvêa C.P.L.: Personal communication (2014).
17. Gouvêa C.P.L., López J.: Software implementation of pairing-based cryptography on sensor networks using the MSP430 microcontroller. In: Sendrier N., Roy B. (eds.) Progress in Cryptology—INDOCRYPT 2009. Lecture Notes in Computer Science, vol. 5922, pp. 248–262. Springer, Berlin (2009). http://conradoplg.cryptoland.net/files/2010/12/indocrypt09.pdf.
18. Gouvêa C.P.L., Oliveira L.B., López J.: Efficient software implementation of public-key cryptography on sensor networks using the MSP430X microcontroller. J. Cryptogr. Eng. **2**(1) (2012). http://conradoplg.cryptoland.net/files/2010/12/jcen12.pdf.
19. Guajardo J., Blümel R., Krieger U., Paar C.: Efficient implementation of elliptic curve cryptosystems on the TI MSP430x33x family of microcontrollers. In: Kim K (ed.) Public Key Cryptography—PKC 2001. Lecture Notes in Computer Science, vol. 1992, pp. 365–382. Springer, Berlin (2001). http://www.emsec.rub.de/media/crypto/veroeffentlichungen/2011/01/21/guajardopkc2001_msp430.pdf.
20. Gura N., Patel A., Wander A., Eberle H., Shantz S.C.: Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In: Joye M (ed.) Cryptographic Hardware and Embedded Systems—CHES 2004. Lecture

Notes in Computer Science, vol. 3156, pp. 119–132. Springer, Berlin (2004). www.iacr.org/archive/ches2004/31560117/31560117.pdf.
21. Hinterwälder G., Moradi A., Hutter M., Schwabe P., Paar C.: Full-size high-security ECC implementation on MSP430 microcontrollers. In: Third International Conference on Cryptology and Information Security in Latin America—Latincrypt 2014. Lecture Notes in Computer Science. Springer, Berlin (2014). http://www.emsec.rub.de/research/publications/Curve25519MSPLatin2014/.
22. Hutter M., Wenger E.: Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In: Preneel, B., Takagi T. (eds.) Cryptographic Hardware and Embedded Systems—CHES 2011. Lecture Notes in Computer Science, vol. 6917, pp. 459–474. Springer, Berlin (2011). http://mhutter.org/papers/Hutter2011FastMultiPrecision.pdf.
23. Hutter M., Schwabe P.: NaCl on 8-bit AVR microcontrollers. In: Youssef A., Nitaj A. (eds.) Progress in Cryptology—AFRICACRYPT 2013. Lecture Notes in Computer Science, vol. 7918, pp. 156–172. Springer, Berlin (2013). http://cryptojedi.org/papers/#avrnacl.
24. Hutter M., Schwabe P.: Multiprecision multiplication on AVR revisited (2014). http://cryptojedi.org/papers/#avrmul.
25. Kenny P.: Formal request from TLS WG to CFRG for new elliptic curves. Posting to the CFRG mailing list (2014). http://www.ietf.org/mail-archive/web/cfrg/current/msg04655.html.
26. Koblitz N.: Elliptic curve cryptosystems. Math. Comput. **48**(177), 203–209 (1987). http://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866109-5/S0025-5718-1987-0866109-5.pdf.
27. Liu A., Ning P.: TinyECC: a configurable library for elliptic curve cryptography in wireless sensor networks. In: International Conference on Information Processing in Sensor Networks—IPSN 2008(April), pp. 22–24, 2008. St. Louis, Missouri, USA, Proceedings, pp. 245–256 (2008). http://discovery.csc.ncsu.edu/pubs/ipsn08-TinyECC-IEEE.pdf.
28. Liu Z., Seo H., Großschädl J., Kim H.: Efficient implementation of NIST-compliant elliptic curve cryptography for sensor nodes. In: Qing S., Zhou J., Liu D. (eds.) Information and Communications Security. Lecture Notes in Computer Science, vol. 8233, pp. 302–317. Springer, Berlin (2013). http://orbilu.uni.lu/bitstream/10993/12934/1/ICICS2013.pdf.
29. Liu Z., Großschädl J., Wenger E.: MoTE-ECC: energy-scalable elliptic curve cryptography for wireless sensor networks. In: Applied Cryptography and Network Security. Lecture Notes in Computer Science, vol. 8479, pp. 361–379. Springer, Berlin (2014). https://online.tugraz.at/tug_online/voe_main2.getvolltext?pCurrPk=77985.
30. Lochter M., Merkle J.: Elliptic curve cryptography (ECC) Brainpool standard curves and curve generation. IETF Request for Comments 5639 (2010). http://tools.ietf.org/html/rfc5639.
31. Miller V.S.: Use of elliptic curves in cryptography. In: Williams H.C. (ed.) Advances in Cryptology—CRYPTO '85: Proceedings. Lecture Notes in Computer Science, vol. 218, pp. 417–426. Springer, Berlin (1986).
32. Montgomery P.L.: Speeding the Pollard and elliptic curve methods of factorization. Math. Comput. **48**(177), 243–264 (1987). http://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866113-7/S0025-5718-1987-0866113-7.pdf.
33. National Institute of Standards and Technology. FIPS PUB 186–4 digital signature standard (DSS) (2013). http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.
34. NXP. LPC1110/11/12/13/14/15 32-bit ARM Cortex-M0 microcontroller; up to 64 kB flash and 8 kB SRAM. Product data sheet, rev. 9.2 edition (2014). http://www.nxp.com/documents/data_sheet/LPC111X.pdf.
35. Pendl C., Pelnar M., Hutter M.: Elliptic curve cryptography on the WISP UHF RFID tag. In: Juels A., Paar C. (eds.) 8th Workshop on RFID Security and Privacy—RFIDsec 2012. Lecture Notes in Computer Science, vol. 7055, pp. 32–47. Springer, Berlin (2012). http://mhutter.org/papers/Pendl2011EllipticCurveCryptography.pdf.
36. ProcFig0. Public key cryptographic algorithm SM2 based on elliptic curves. Part 1: General. (2012). http://www.oscca.gov.cn/UpFile/20101222214822692.pdf.
37. Scott M.: Re: NIST announces set of elliptic curves. Posting to the sci.crypt mailing list (1999). https://groups.google.com/forum/message/raw?msg=sci.crypt/mFMukSsORmI/FpbHDQ6hM_MJ.
38. Seo H., Kim H.: Multi-precision multiplication for public-key cryptography on embedded microprocessors. In: Lee D.H., Yung M (eds.) Information Security Applications. Lecture Notes in Computer Science, vol. 7690, pp. 55–67. Springer, Berlin (2012). doi:10.1007/978-3-642-35416-8
39. Seo H., Kim H.: Optimized multi-precision multiplication for public-key cryptography on embedded microprocessors. Int. J. Comput. Commun. Eng. **2**(3), (2013). http://www.ijcce.org/papers/183-J034.pdf.
40. Szczechowiak P., Oliveira L.B., Scott M., Collier M., Dahab R.: NanoECC: testing the limits of elliptic curve cryptography in sensor networks. In: Verdone R. (ed.) Wireless Sensor Networks. Lecture Notes

in Computer Science, vol. 4913, pp. 305–320. Springer, Berlin (2008). http://www.ic.unicamp.br/~leob/publications/ewsn/NanoECC.pdf.
41. Texas Instruments Incorporated. MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx family user's guide (2012). www.ti.com.cn/cn/lit/ug/slau367f/slau367f.pdf.
42. Texas Instruments Incorporated. MSP430x2xx family user's guide (2004). http://www.ti.com/lit/ug/slau144j/slau144j.pdf.
43. Texas Instruments Incorporated. MSP-EXP430FR5969 LaunchPad Development Kit user's guide (2014). http://www.ti.com/lit/ug/slau535a/slau535a.pdf.
44. Wenger E., Werner M.: Evaluating 16-bit processors for elliptic curve cryptography. In: Prouff E. (ed.) Smart Card Research and Advanced Applications—CARDIS 2011. Lecture Notes in Computer Science, vol. 7079, pp. 166–181. Springer, Berlin (2011). https://online.tugraz.at/tug_online/voe_main2.getvolltext?pCurrPk=59062.
45. Wenger E., Unterluggauer T., Werner M.: 8/16/32 shades of elliptic curve cryptography on embedded processors. In: Paul G., Vaudenay S. (eds.) Progress in Cryptology—INDOCRYPT 2013. Lecture Notes in Computer Science, vol. 8250, pp. 244–261. Springer, Berlin (2013). https://online.tugraz.at/tug_online/voe_main2.getvolltext?pCurrPk=72486.