

# Full-Size High-Security ECC Implementation on MSP430 Microcontrollers

Gesine Hinterwlder<sup>1,2(✉)</sup>, Amir Moradi<sup>1</sup>, Michael Hutter<sup>3</sup>, Peter Schwabe<sup>4</sup>,  
and Christof Paar<sup>1,2</sup>

<sup>1</sup> Horst Grtz Institute for IT Security, Ruhr-University Bochum, Bochum, Germany  
{gesine.hinterwaelder, amir.moradi, christof.paar}@rub.de

<sup>2</sup> Department of Electrical and Computer Engineering, University of Massachusetts  
Amherst, Amherst, USA

<sup>3</sup> Institute for Applied Information Processing and Communications (IAIK),  
Graz University of Technology, Graz, Austria  
Michael.Hutter@iaik.tugraz.at

<sup>4</sup> Digital Security Group, Radboud University Nijmegen, Nijmegen, The Netherlands  
peter@cryptojedi.org

**Abstract.** In the era of the Internet of Things, smart electronic devices facilitate processes in our everyday lives. Texas Instrument’s MSP430 microcontrollers target low-power applications, among which are wireless sensor, metering and medical applications. Those domains have in common that sensitive data is processed, which calls for strong security primitives to be implemented on those devices. Curve25519, which builds on a 255-bit prime field, has been proposed as an efficient, highly-secure elliptic-curve. While its high performance on powerful processors has been shown, the question remains, whether it is suitable for use in embedded devices. In this paper we present an implementation of Curve25519 for MSP430 microcontrollers. To combat timing attacks, we completely avoid conditional jumps and loads, thus making our software constant time. We give a comprehensive evaluation of different implementations of the modular multiplication and show which ones are favorable for different conditions. We further present implementation results of Curve25519, where our best implementation requires 9.1 million or 6.5 million cycles on MSP430Xs having a  $16 \times 16$ -bit or a  $32 \times 32$ -bit hardware multiplier respectively.

**Keywords:** MSP430 · Carry-save representation · Karatsuba · Operand-caching multiplication · Curve25519

---

\* This work was supported in part by the German Federal Ministry for Economic Affairs and Energy (Grant 01ME12025 SecMobil), by the Netherlands Organisation for Scientific Research (NWO) through Veni 2013 project 13114, and by the Austrian Science Fund (FWF) under the grant number TRP251-N23. Permanent ID of this document: 0b3f1ea83d48e400ad1def71578c4c66. Date: 2014-10-01.

# 1 Introduction

Implantable medical devices execute services essential for a patient’s well-being. Their power consumption must be very low, as they operate either entirely based on harvested power, or contain a battery, which can only be replaced by surgery. Many of them communicate wirelessly over an RF channel, which allows for configuration of those devices without surgical intervention. However, the wireless channel also poses potential attack possibilities, as shown by Halperin et al. in [12]. This calls for strong security mechanisms to be implemented on those very constrained devices.

Texas Instruments designed MSP430 microcontrollers to target low-power applications, and advertises the application of MSP430s in the domain of medical devices [16]. MSP430s can be operated at low voltages (1.8 to 3.3 V). Newer devices of the MSP430 family have AES hardware accelerators that support 256-bit AES. Yet, many security services that are desirable for wireless communication, especially in the domain of medical devices, rely on public-key cryptography. This naturally raises the question about the performance of public-key cryptography on MSP430 microcontrollers.

Bernstein introduced the Curve25519 elliptic-curve Diffie-Hellman key exchange protocol in 2006 [2]. It uses a Montgomery curve defined over a 255-bit prime field and achieves a security level of 128 bits. Montgomery curves are known to allow for very efficient variable-base-point single-scalar multiplication, which makes this curve attractive for elliptic-curve key-agreement schemes.

**Our Contribution.** In this paper, we present a full implementation of the Curve25519 Diffie-Hellman key-agreement scheme on MSP430X microcontrollers<sup>1</sup>. We differentiate those MSP430Xs with a  $16 \times 16$ -bit and those with a  $32 \times 32$ -bit hardware multiplier and developed our code for both platforms. As all previous implementations of Curve25519, we use projective coordinates for the elliptic-curve point representation. The main performance bottleneck of the variable-base-point single-scalar multiplication are thus modular multiplications in the underlying prime field. We hence put our focus on optimizing the modular multiplication on the MSP430 architecture, and give a comprehensive evaluation of different implementation techniques for MSP430 microcontrollers.

We use the Montgomery powering ladder [24] to implement the scalar multiplication on the elliptic curve, since this is a highly regular algorithm, making the executed computation independent of the scalar. Our software completely avoids input-dependent loads and branches, thus executing in constant time and thus inherently protecting against timing attacks such as [1] or [31].

We evaluate our implementation by executing it on Texas Instrument’s MSP-EXP430FR5969 LaunchPad Evaluation Kit. This board integrates an MSP430-FR5969 microcontroller [28] with a  $32 \times 32$ -bit hardware multiplier, which is built into the WISP 5.0 UHF computational RFID tag<sup>2</sup>, a device that operates

<sup>1</sup> The software is available at <http://emsec.rub.de/research/publications/Curve25519MSPLatin2014/>.

<sup>2</sup> <http://wisp.wikispaces.com/WISP%205.0>.

based on harvested power from the RF field. With a price of a few dollars, this microcontroller is a suitable target for wireless sensor and medical applications.

**Related Work.** Curve25519 has been implemented on several platforms. In the paper introducing Curve25519 [2], Bernstein presented implementation results for several Intel Pentium and an AMD Athlon processor. In 2009, Costigan and Schwabe presented Curve25519 software for the Cell Broadband Engine [7]. In 2012, Bernstein and Schwabe presented an implementation for ARM processors with NEON vector instructions [5]. Recently, Sasdrich and Güneysu presented an implementation on reconfigurable hardware in [26]. Another recent publication shows an implementation of Curve25519, that fits into 18 tweets [6, 20]. So far, only one implementation shows performance results of Curve25519 on constrained devices, namely the implementation for 8-bit AVR microcontrollers by Hutter and Schwabe presented in [13]. No previous work has yet shown implementation results of Curve25519 for 16-bit microcontrollers.

There exist many publications on Elliptic Curve Cryptography (ECC) implementations on the MSP430 microcontroller architecture. One of the first publications of asymmetric cryptography on the MSP430 is by Guajardo, Blümel, Krieger, and Paar in 2001 [11]. They presented an implementation of an elliptic curve with a security level of 64 bits and show that a scalar multiplication can be performed within 3.4 million clock cycles. In 2007, Scott and Szczechowiak presented optimizations for underlying ECC finite-field multiplications [27]. Their  $160 \times 160$ -bit (hybrid) multiplication method requires 1746 cycles. In 2009, Szczechowiak, Kargl, Scott, and Collier presented pairing-based cryptography on the MSP430 [29]. Similar results have been reported by Gouvêa and López in the same year [9]. They reported new speed records for 160-bit and 256-bit finite-field multiplications on the MSP430 needing 1586 and 3597 cycles, respectively. They further presented an implementation of a 256-bit elliptic curve random scalar multiplication needing 20.4 million clock cycles. In 2011, Wenger and Werner compared ECC scalar multiplications on various 16-bit microcontrollers [33]. Their Montgomery-ladder based scalar multiplication needs 23.9 million cycles using a NIST P-256 elliptic curve. Also in 2011, Pendl, Pelnar, and Hutter presented the first ECC implementation running on the WISP UHF RFID tag [25]. Their 192-bit NIST curve implementation achieves an execution time of around 10 million clock cycles. They also reported first multi-precision multiplication results for 192 bits needing 2581 cycles. In 2012, Gouvêa, Oliveira, and López reported new speed records for different MSP430 architectures. They improved their results from [9], namely, for the MSP architecture (with a  $16 \times 16$  multiplier) their 160-bit and 256-bit finite-field multiplication implementations need 1565 and 3563 cycles, respectively.

Also note that there exist recent works to extend the MSP430 with instruction-set extensions. In 2013, Wenger, Unterluggauer, and Werner [32] presented an MSP430 clone in hardware that implements a special instruction-set extension. For a NIST P-256 elliptic curve, their Montgomery ladder implementation requires 9 million clock cycles – without instruction-set extensions (and to put these numbers in relation), their implementation needs 22.2 million cycles.

There also exist several software libraries for the MSP430 that support ECC. These libraries mainly target sensor nodes such as the Tmote Sky which are equipped with an MSP430 microcontroller. Examples are the NanoECC [30], TinyECC [22], and MIRACL [23] libraries, and the RELIC toolkit [8].

Under the common assumption that the execution time of ECC grows approximately as a cubic function of the field size, our software significantly outperforms all presented ECC implementations on MSP430 microcontrollers in speed, while executing in constant time, thus providing security against timing attacks.

**Organization.** Section 2 describes specifics about the MSP430 architecture important for our implementation. Section 3 describes general basics about the implementation of Curve25519, Sect. 4 presents a detailed description of the various implementation techniques for modular multiplications that we investigated. Implementation and measurement results are presented in Sect. 5, and we conclude our work with Sect. 6.

## 2 The MSP430X Microcontroller Architecture

We implemented the modular multiplication operation for MSP430X devices that feature a  $16 \times 16$ -bit hardware multiplier as well as for those that feature a  $32 \times 32$ -bit multiplier, and show which implementation technique is preferable on either platform. We give cycle count estimations for the MSP430F2618 [19], which has a  $16 \times 16$ -bit hardware multiplier, and cycle count estimations as well as execution results for the MSP430FR5969 [28], which has a  $32 \times 32$ -bit hardware multiplier. But, our results can be generalized to other microcontrollers from the MSP430 family. This section describes specifics about the MSP430X architecture that are important for the discussion of the implementation techniques. For more details about the MSP430X architecture, we refer the reader to the MSP430x2xx user’s guide [18].

**Processing Unit.** Both MSP430 microcontrollers that we consider have a 16-bit RISC CPU, with 27 core instructions and 24 emulated instructions. The CPU has 16-bit registers, of which R0 to R3 are special-purpose registers and R4 to R15 are freely usable working registers. The execution time of all register operations is one cycle, but the overall execution time for an instruction depends on the instruction format and the addressing mode.

**Addressing Mode.** The CPU features 7 addressing modes. Our implementation uses the register mode, indexed mode, absolute mode, indirect auto-increment mode, and immediate mode. It is important to note that while indirect auto-increment mode saves one clock cycle on all operations compared to indexed mode, only indexed mode can be used to store results back to RAM.

**Hardware Multiplier.** Both devices that we consider feature memory-mapped hardware multipliers, which work 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 are peripheral

registers, which have to be loaded with CPU instructions. The result is stored in two (in case of  $16 \times 16$ -bit multipliers) or four (in case of  $32 \times 32$ -bit multipliers) 16-bit registers. A register `SUMEXT` is available, which is similar to the status register in the main CPU. This register shows for the multiply-and-accumulate instructions, whether a multiplication has produced a carry bit. It is not possible to accumulate carries in `SUMEXT`. The time that is required for the multiplication is determined by the time it takes to load the multiplier registers.

### 3 Implementation of Curve25519

Curve25519 is an elliptic curve in Montgomery form. This curve has been carefully chosen to provide very high performance for Diffie-Hellman key agreement at the 128-bit security level. It is defined by the equation  $y^2 = x^3 + 486662x^2 + x$  over the prime field  $\mathbb{F}_{2^{255}-19}$ . For details about the choice of curve and security see [2].

The key-agreement scheme computes a 32-byte shared secret  $Q_x$  from a 32-byte secret key  $n$  and a 32-byte public key  $P_x$ . Here  $Q_x$  and  $P_x$  are  $x$ -coordinates of points on the elliptic curve. At its core, the Curve25519 Diffie-Hellman key-agreement scheme executes a variable-base-point single-scalar multiplication on the elliptic curve, multiplying the public key  $P_x$  with the secret key  $n$ , to obtain the shared secret  $Q_x$ . Special conditions are given for the secret scalar  $n$ , namely that the 3 least significant bits and the most significant bit are set to zero, and the second-most significant bit is set to 1 [4].

We follow the suggestions of [2] for implementing the variable-base-point single-scalar multiplication on the elliptic curve. We used the Montgomery powering ladder [24] of 255 “ladder steps”. Each ladder step computes a differential point addition and a point doubling. Starting with the points `R1` and `R2`, in each ladder step either `R2` is added to `R1` ( $R1 \leftarrow R1 + R2$ ) and then `R2` is doubled ( $R2 \leftarrow 2 \cdot R2$ ), or `R1` is added to `R2` ( $R2 \leftarrow R2 + R1$ ) and then `R1` is doubled ( $R1 \leftarrow 2 \cdot R1$ ). To avoid conditional load addresses that can lead to cache-timing attacks, we execute the same operations ( $R1 \leftarrow R1 + R2$  and  $R2 \leftarrow 2 \cdot R2$ ) in each iteration, and swap the contents of `R1` and `R2` depending on the scalar bit  $b$ .

Note that for the conditional swap we do not use branch instructions. Instead, this operation is implemented as follows: An unsigned variable  $\hat{b}$  is cleared. Then  $b$  is subtracted from  $\hat{b}$  leading to  $\hat{b}$  being 0 or 0xffff, depending on whether  $b$  is 0 or 1. To swap the contents of  $x$  and  $y$ , an auxiliary variable is used to store  $t_{swp} = x \oplus y$ .  $t_{swp}$  is anded with the value stored in  $\hat{b}$ , resulting in  $t_{swp} = x \oplus y$  for  $b = 1$  and  $t_{swp} = 0$  otherwise. Then  $t_{swp}$  is xored with  $x$  and  $y$  leading to either the original values being stored in  $x$  and  $y$  for  $b = 0$ , or the swapped values for the case of  $b = 1$ . Together with the constant-time field arithmetic we thus obtain a fully timing-attack protected constant-time implementation.

In [24] Montgomery presented  $x$ -coordinate-only doubling and differential-addition formulas for points on a curve defined by an equation of the form  $By^2 = x^3 + Ax^2 + x$ . He showed the correctness of those formulas, which rely on standard-projective-coordinate representation of the points, for the case of inputs not being equal to the point at infinity. In [2] Bernstein extended the proof of correctness

---

**Algorithm 1.**  $x$ -coordinate-only variable base-point single-scalar point multiplication on Curve25519 based on the Montgomery powering ladder [2, 7].

---

**Input** :  $n \in \mathbb{Z}$ ,  $P_x$ ,  $x$ -coordinate of point  $P$ .

**Output**:  $Q_x$ ,  $x$ -coordinate of point  $Q \leftarrow n \cdot P$ .

---

```

1  $X_1 \leftarrow P_x; X_2 \leftarrow 1; Z_2 \leftarrow 0; X_3 \leftarrow P_x; Z_3 \leftarrow 1$ 
2 for  $i = 254$  downto 0 do
3   if  $n_i \neq n_{i-1}$  then
4      $\text{swap}(X_2, X_3)$       /* This conditional swapping is implemented */
5      $\text{swap}(Z_2, Z_3)$       /* in constant time (see Sect.3). */
6   end
7    $t_1 \leftarrow X_2 + Z_2$ 
8    $t_2 \leftarrow X_2 - Z_2$ 
9    $t_3 \leftarrow X_3 + Z_3$ 
10   $t_4 \leftarrow X_3 - Z_3$ 
11   $t_6 \leftarrow t_1^2$ 
12   $t_7 \leftarrow t_2^2$ 
13   $t_5 \leftarrow t_6 - t_7$ 
14   $t_8 \leftarrow t_4 \cdot t_1$ 
15   $t_9 \leftarrow t_3 \cdot t_2$ 
16   $X_3 \leftarrow (t_8 + t_9)^2$ 
17   $Z_3 \leftarrow X_1(t_8 - t_9)^2$ 
18   $X_2 \leftarrow t_6 \cdot t_7$ 
19   $Z_2 \leftarrow t_5(t_7 + 121666t_5)^2$ 
20 end
21 if  $n_0 == 1$  then
22    $\text{swap}(X_2, X_3)$       /* This conditional swapping is implemented */
23    $\text{swap}(Z_2, Z_3)$       /* in constant time (see Sect.3). */
24 end
25  $Z_2 \leftarrow 1/Z_2$ 
26 return  $(X_2 \cdot Z_2)$ 

```

---

to the case of an input being equal to the point at infinity. Using these formulas, a differential addition of two points requires 4 multiplications and 2 squarings. Point doubling requires 2 multiplications, 2 squarings, and one multiplication by the constant  $(486662 + 2)/4 = 121666$ . The differential-addition formula requires as input the difference of the input points. If the  $Z$ -coordinate of this difference point is one, the addition formula can be reduced to require only 3 multiplications and 2 squarings. Algorithm 1 summarizes the  $x$ -coordinate-only variable-base-point single-scalar point multiplication on Curve25519 requiring 255 differential additions and doublings (ladder steps), 255 conditional swaps, and one inversion at the end to transform the result back to affine coordinates [2, 7].

## 4 Implementation of Modular Multiplication in $\mathbb{F}_{2^{255}-19}$

Many techniques have been proposed to improve the performance of multi-precision multiplication implementations, especially for constrained devices. In the following we describe which techniques we implemented for the MSP430X architecture. To have a fair comparison, all methods were implemented in assembly and were fully unrolled.

**Representation of Big Integers.** We use an unsigned radix-2<sup>16</sup> representation for the operand-caching [15] and the Karatsuba multiplication [14, 21], and a signed radix-2<sup>[255/26]</sup> representation for the carry-save implementation. In unsigned radix-2<sup>16</sup> representation, an  $n$ -bit integer  $A$  is represented as an array of  $m = \lceil n/16 \rceil$  words in little-endian order as  $(a_0, a_1, \dots, a_{m-1})$ , such that  $A = \sum_{i=0}^{m-1} a_i 2^{16i}$  where  $a_i \in \{0, \dots, 2^{16} - 1\}$ . In the radix-2<sup>[255/26]</sup> representation an  $n$ -bit integer  $B$  is represented as an array of  $\ell = \lceil 26n/255 \rceil$  16-bit words in little-endian order as  $(b_0, b_1, \dots, b_{\ell-1})$ , such that  $B = \sum_{j=0}^{\ell-1} b_j 2^{\lceil 255j/26 \rceil}$ , where  $b_j \in \{-2^{15}, \dots, 2^{15} - 1\}$ . Hence, in the radix-2<sup>[255/26]</sup> representation an element in  $\mathbb{F}_{2^{255}-19}$  is represented using 26 16-bit words. Since inputs and outputs to the scalar multiplication on Curve25519 are 32-byte arrays, conversions to and from the used representations are executed at the beginning and the end of the complete scalar multiplication.

### 4.1 Multiplication Using Carry-Save Representation

This implementation follows the fast arithmetic implementation presented in [2]. An integer is represented using the signed radix-2<sup>[255/26]</sup> representation. Beneficial of this representation is that an addition or subtraction can be executed without having to consider carry bits. It only requires pairwise addition or subtraction of the respective coefficients, as long as the result of coefficient additions or subtractions does not exceed the word-length. An element in this representation looks as follows:

$$B = b_0 + b_1 2^{10} + b_2 2^{20} + b_3 2^{30} + b_4 2^{40} + b_5 2^{50} + b_6 2^{59} + b_7 2^{69} + b_8 2^{79} + \dots + b_{25} 2^{246}.$$

Figure 1 presents the steps executed to compute the first 8 coefficients  $r_i$  of the multiplication  $r \leftarrow f \times g$ . After transforming an integer to radix-2<sup>[255/26]</sup>

...	$r_7$	$r_6$	$r_5$	$r_4$	$r_3$	$r_2$	$r_1$	$r_0$
...	$f_7 g_0$	$f_6 g_0$	$f_5 g_0$	$f_4 g_0$	$f_3 g_0$	$f_2 g_0$	$f_1 g_0$	$f_0 g_0$
...	$f_7 g_1$	$2 f_6 g_1$	$f_4 g_1$	$f_3 g_1$	$f_2 g_1$	$f_1 g_1$	$f_0 g_1$	$38 f_{24} g_2$
...	$2 f_6 g_2$	$2 f_4 g_2$	$f_3 g_2$	$f_2 g_2$	$f_1 g_2$	$f_0 g_2$	$38 f_{23} g_3$	$38 f_{23} g_3$
...	$2 f_4 g_3$	$2 f_3 g_3$	$f_2 g_3$	$f_1 g_3$	$f_0 g_3$	$38 f_{25} g_4$	$38 f_{24} g_4$	$38 f_{22} g_4$
...	$2 f_3 g_4$	$2 f_2 g_4$	$f_1 g_4$	$f_0 g_4$	$38 f_{23} g_4$	$38 f_{24} g_4$	$38 f_{23} g_4$	$38 f_{21} g_5$
...	$2 f_2 g_5$	$2 f_1 g_5$	$f_0 g_5$	$38 f_{25} g_5$	$38 f_{24} g_5$	$38 f_{23} g_5$	$38 f_{22} g_5$	$38 f_{20} g_6$
...	$f_1 g_6$	$f_0 g_6$	$19 f_{25} g_6$	$19 f_{24} g_6$	$19 f_{23} g_6$	$19 f_{22} g_6$	$19 f_{21} g_6$	$38 f_{19} g_7$
...	$f_0 g_7$	$19 f_{25} g_7$	$19 f_{24} g_7$	$19 f_{23} g_7$	$19 f_{22} g_7$	$19 f_{21} g_7$	$38 f_{20} g_7$	$38 f_{18} g_8$
...	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

**Fig. 1.** Visualisation computation of coefficients for carry-save multiplication.

representation, each coefficient  $b_i$  of  $B$  is within  $(-2^9, 2^9)$  or  $(-2^{10}, 2^{10})$ . We precompute  $2f$  and  $19g$  to easily realize constant multiplication with factors 2, 19, and 38. We use the product-scanning technique to compute the coefficients  $r_i$ , interleaving the multiplication with the reduction, i.e., we compute a coefficient and reduce it right away. For the computation of each  $r_i$ , 26 products of coefficients have to be added.

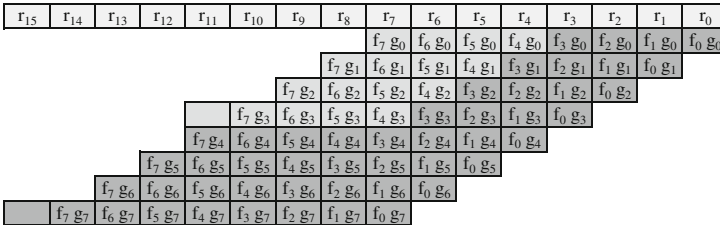
This type of implementation has two disadvantages on the MSP430X architecture. First of all the MSP430 has very few general-purpose registers, while the inputs have to be loaded from four different arrays  $f, g, 2f$  and  $19g$ . This makes storing inputs in registers difficult, as different operands are loaded for computation of the various coefficients. Further, while we use indirect auto-increment mode to access  $g$  and  $19g$ , there is no indirect auto-decrement mode on the MSP430 and we need to access the other inputs using the costly indexed mode. The other disadvantage is the highly complex reduction of a coefficient, requiring several shift operations, which are expensive on MSP430 devices.

Since we could not achieve good performance results with this type of implementation, we tried to speed things up relying on the refined Karatsuba formulas presented in [3]. A problem occurs when trying to add the low and the high part of  $B$  in signed radix- $2^{\lceil 255/26 \rceil}$  representation. For example computing the coefficient of  $2^{40}$  cannot be done by adding  $b_4$  and  $b_{16}$  as  $b_{16}$  would be input to exponent  $2^{39}$ . Our solution to this was to represent elements using signed radix- $2^{\lceil 256/26 \rceil}$  representation and rely on computations modulo  $2^{256} - 38$ . Yet still, the disadvantages of this type of implementation on the MSP430 architecture dominate the advantages.

## 4.2 Operand-Caching Multiplication

Operand-caching was proposed by Hutter and Wenger in 2011 [15]. The idea of this method is to reduce the number of load instructions by organizing the operations in a way that allows the same input operands to be used for multiple computations.

Figure 2 shows a toy-size example of the operand-caching multiplication. Here the execution of computations is divided into the light gray and the dark gray area. First the light gray block is computed followed by the dark gray area.



**Fig. 2.** Visualisation of the operand-caching method for 2 elements consisting of 8 words.

The empty dark gray and light gray boxes represent space that is required for carry-bits.

As we have 8 general-purpose registers available for storing operands during the execution of the multiplication, we chose the row size to be 4. Since each input array has 16 elements,  $16/4 = 4$  rows have to be computed. Many loads to the hardware multiplier can be saved when loading operands in a special order. For each operation of the hardware multiplier OP2 has to be loaded to start execution. Yet, MAC does not have to be loaded each time. If it is not loaded, it uses the value that had been loaded to MAC in the previous use of the hardware multiplier. For example, if for the computation of  $r_1$ , as the final step  $f_0$  was loaded to MAC and  $g_1$  to OP2, then we start the computation of  $r_2$  by loading  $g_2$  to OP2.

In this multiplication we first multiply both inputs  $f$  and  $g$ , resulting in a double-sized array and then reduce this result. Since reducing mod  $2^{255} - 19$  requires bit shifts, we chose to reduce intermediate results mod  $2^{256} - 38$  and only reduce the final result mod  $2^{255} - 19$ . We implemented two versions of operand-caching multiplication, one making use of the  $32 \times 32$ -bit hardware multiplier (in the following called 32-bit operand-caching) and the other only loading 16-bit inputs to the multiplier (in the following called 16-bit operand-caching). Naturally the implementation that makes use of the  $32 \times 32$ -bit hardware multiplier is faster and also requires less code space, since fewer loads to the multiplier have to be performed.

### 4.3 Karatsuba Multiplication

This section is based on a very recent paper on the implementation of multi-precision multiplication on AVR microcontrollers [14]. Karatsuba presented a sub-quadratic multiplication method that reduces the number of required word multiplications for multi-precision multiplications [21]. The implementation by Hutter and Schwabe [14] is based on this idea and first demonstrates that this method is more advisable on AVRs even for very small input sizes starting from 48 bits. They implemented what they call *subtractive Karatsuba*. This method avoids having to take extra carry bits into account by computing  $|F_l - F_h|$  and  $|G_l - G_h|$  instead of  $F_l + F_h$  and  $G_l + G_h$ , which makes it easier to obtain a constant-time implementation. In the following we report the method, as it was presented in [14], adapting it to the case of a 16-bit architecture. The steps for multiplying two  $n$ -byte numbers, where in our case  $n = 32$ , are described in detail. Using a 16-bit architecture, we have to process arrays of  $n/2 = 16$  elements. We split those arrays at  $k = 16/2 = 8$ .

- Write  $F = F_\ell + 2^{16k} F_h$  and  $G = G_\ell + 2^{16k} G_h$
- compute  $L = F_\ell \cdot G_\ell$
- compute  $H = F_h \cdot G_h$
- compute  $M = |F_\ell - F_h| \cdot |G_\ell - G_h|$  and
- set  $t = 0$ , if  $M = (F_\ell - F_h) \cdot (G_\ell - G_h)$ ;  $t = 1$  otherwise;
- compute  $\hat{M} = (-1)^t M$ ; and
- obtain the result as  $FG = L + 2^{16k}(L + H - \hat{M}) + 2^{16n/2}H$ .

We use operand-caching multiplication for all multi-precision multiplications within the Karatsuba multiplication, i.e., the computations of  $L$ ,  $H$ , and  $M$ .  $|F_\ell - F_h|$  is computed as follows: first we subtract with borrow all elements in  $F_h$  from those in  $F_\ell$  and subtract with borrow from a register  $b_F$  that was cleared before. This results in  $b_F = 0$  for  $F_\ell > F_h$  and  $b_F = 0\text{xffff}$  otherwise. We XOR  $b_F$  with  $F_\ell - F_h$  resulting in the ones-complement of  $F_\ell - F_h$ . We then compute  $t_F = b_F$  AND 1 add this to the ones-complement of  $F_\ell - F_h$  and ripple the carry through, resulting in the two’s complement of  $F_\ell - F_h$ , which is equal to  $|F_\ell - F_h|$ .  $|G_\ell - G_h|$  is computed similarly. The value  $t$  required for the computation of  $M$  is obtained as  $t = t_F \oplus t_G$ . The same technique that was used to compute the absolute difference above is used for the computation of  $\hat{M}$  from  $M$ , leaving out the initial subtraction part.

Again we computed the product of the inputs resulting in a double-sized array and reduced the result mod  $2^{256} - 38$ . Only at the end of the Curve25519 computation we reduced results mod  $2^{255} - 19$ . In the following we will refer to the implementation making use of the  $32 \times 32$ -bit multiplier as 32-bit Karatsuba and the one for  $16 \times 16$ -bit multiplier as 16-bit Karatsuba. We further implemented this method for 2-level Karatsuba, i.e. using subtractive Karatsuba for the computation of  $L$ ,  $H$ , and  $M$ . We will refer to those implementations as 2-Level 32-bit Karatsuba and 2-Level 16-bit Karatsuba, for using  $32 \times 32$ -bit multiplier and  $16 \times 16$ -bit multiplier respectively.

## 5 Performance and Power Consumption Results

We used IAR Embedded Workbench for MSP430 IDE version 5.60.3 to develop our code and compiled all source code by setting the compiler options to “low”. This causes dead code, redundant labels and redundant branches to be eliminated and achieves that variables live only as long as they are needed. It further avoids common subexpression elimination, loop unrolling, function inlining, code motion and type-based alias analysis [17]. Note that all functions implementing arithmetic in  $\mathbb{F}_{2^{255}-19}$  were implemented in assembly, while the higher level functions are implemented in C. This section describes our implementation and measurement results.

We first present cycle-count estimates for the modular multiplication implementations given by IAR Embedded Workbench IDE. We compare these results for two devices, namely MSP430FR5969 and MSP430F2618 having a  $32 \times 32$ -bit and a  $16 \times 16$ -bit hardware multiplier, respectively. We further present numbers for the required code space for the multiplication implementations.

For a device that has a  $32 \times 32$ -bit hardware multiplier (MSP430FR5969) we executed the code and measured the execution time using the debugging functionality of IAR Embedded Workbench IDE. We present the cycle count for an execution of the Curve25519 variable-base-point single-scalar multiplication on the MSP430FR5969 for the cases of having a  $32 \times 32$ -bit or a  $16 \times 16$ -bit hardware multiplier on this target. Finally, we present our power measurement results of the execution of different multiplication implementations and the scalar multiplication on the MSP-EXP430FR5969 Launchpad Evaluation Kit.

**Table 1.** Simulated cycle count for modular multiplication (including reduction) on MSP430F2618 and MSP430FR5969, given by IAR Embedded Workbench IDE version 5.60.3

		MSP430FR5969	MSP430F2618
1	16-bit Operand-caching	3968	3949
2	32-bit Operand-caching	2505	-
3	16-bit Carry-save	7231	7228
4	16-bit Karatsuba	3666	3623
5	32-bit Karatsuba	2501	-
6	16-bit 2-level Karatsuba	3595	3554
7	32-bit 2-level Karatsuba	2705	-

**Table 2.** Code space (in bytes) required for modular multiplication implementations (including reduction) on MSP430s.

		Code Space (in bytes)
1	16-bit Operand-caching	4762
2	32-bit Operand-caching	2878
3	16-bit Carry-save	8448
4	16-bit Karatsuba	4316
5	32-bit Karatsuba	2826
6	16-bit 2-level Karatsuba	4270
7	32-bit 2-level Karatsuba	3144

## 5.1 Performance

First we simulated the cycle count and measured the required code space of the different variants of implementation of the modular multiplication that we implemented in IAR Embedded Workbench IDE. Table 1 presents the simulated execution times for the two aforementioned microcontrollers, while Table 2 shows the required code space for each implementation. It seems quite natural that the version making use of the  $32 \times 32$ -bit hardware multiplier is faster and requires less code space since fewer load (and store) operations to (and from) the dedicated registers of the multiplier have to be executed.

We then measured the execution time of all multiplication implementations on the MSP430FR5969 using the debugging functionality of IAR Embedded Workbench IDE (Table 3). During this step we realized that wait cycles must be included when the MSP430FR5969 runs at the frequency of 16 MHz. It is due to the limited access frequency of FRAM, i.e., 8 MHz. So, the speed of the implementation is not doubled by increasing the operation frequency from 8 MHz to 16 MHz. Table 3 displays these results. While in simulation the 32-bit operand-caching multiplication seems to perform similar to the 32-bit Karatsuba

implementation, it turns out that, when executing the implementations on the board the 32-bit Karatsuba implementation performs a bit better compared to 32-bit operand-caching (cf. Table 3). This is due to the fact that IAR Embedded Workbench IDE does not correctly simulate the execution time of the hardware multiplier, i.e. the time it takes until the CPU can read out results from the hardware multiplier. Interestingly, the improvement of using 2-level Karatsuba is only given when making use of the  $16 \times 16$ -bit hardware multiplier (MSP430F2618). When making use of the  $32 \times 32$ -bit multiplier, the overhead required for the implementation of 2-level Karatsuba seems to dominate over the improvements in timings. The lowest code space is achieved with 32-bit Karatsuba, but not far from 32-bit operand-caching (Table 2).

**Table 3.** Execution time (i.e., cycle count) on MSP-EXP430FR5969 Launchpad Evaluation Kit, optimizations set to “low” when running the microcontroller at different frequencies.

		8 MHz	16 MHz
1	16-bit operand-caching	4045	4599
2	32-bit operand-caching	2529	2864
3	16-bit Carry-save	7230	8289
4	16-bit Karatsuba	3696	4203
5	32-bit Karatsuba	2488	2824
6	16-bit 2-level Karatsuba	3606	4119
7	32-bit 2-level Karatsuba	2684	3069

Further we implemented the variable-basepoint single-scalar multiplication for the cases of having a  $32 \times 32$ -bit and having a  $16 \times 16$ -bit hardware multiplier. For the implementation that makes use of the  $32 \times 32$ -bit hardware multiplier we used 32-bit Karatsuba and for the implementation that only requires a  $16 \times 16$ -bit hardware multiplier we used 2-level 16-bit Karatsuba, as those are the fastest implementations for those cases according to Table 3. On the MSP430FR5969 the x-coordinate-only variable-basepoint single-scalar multiplication, which makes use of the  $32 \times 32$ -bit hardware multiplier, executes in 6,513,011 clock cycles and requires 9.1 kB of code space, whereas the  $16 \times 16$ -bit hardware multiplier version, executes in 9,139,739 clock cycles and requires 11.6 kB of code space.

Since there are no implementation results of the plain ECC point multiplication on an MSP430X with a  $32 \times 32$ -bit hardware multiplier given in the literature, we compare the results given in the literature to our result for the  $16 \times 16$ -bit hardware multiplier (Table 4). Note that Gouvea et al. obtain better performance results for a 128-bit-secure elliptic-curve scalar multiplication on an MSP430X microcontroller with a  $32 \times 32$ -bit hardware multiplier, albeit on a different curve [10], but do not report performance results for the plain scalar multiplication, but instead for the execution of several ECC-based protocols.

**Table 4.** Execution time (i.e., cycle count) of variable base-point single-scalar multiplications on an elliptic curve providing a security level comparable to 128-bit symmetric security on MSP430 microcontrollers.

	Architecture	Cycle count
Wenger et al. [33]	MSP	23,973,000
Wenger et al. [32]	MSP Clone w/o ISE	22,170,000
Gouvêa et al. [9]	MSP	20,476,234
<b>Our implementation</b>	MSPX	<b>9,139,739</b>

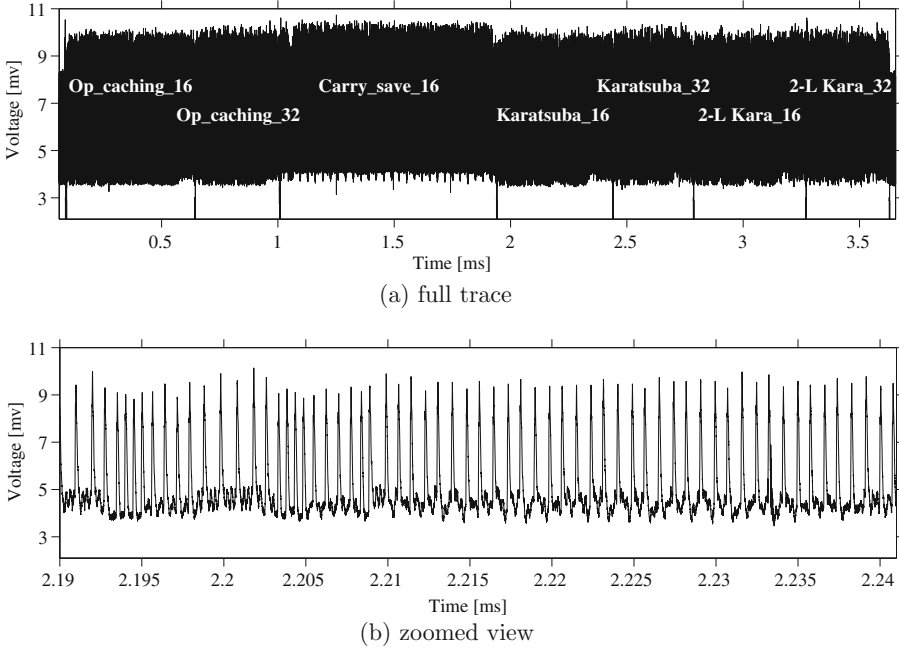
## 5.2 Power Consumption

We further examined our code in terms of power consumption on the MSP-EXP430FR5969 Launchpad Evaluation Kit. We have implemented all multiplications (e.g., listed in Table 1) in such a way that first two random operands are selected then multiplied together by all multiplication algorithms one after another. We also used an I/O pin of the MSP-EXP430FR5969 Launchpad Evaluation Kit to indicate the start and the end of each algorithm thereby being able to identify at which period of time each algorithm is executed.

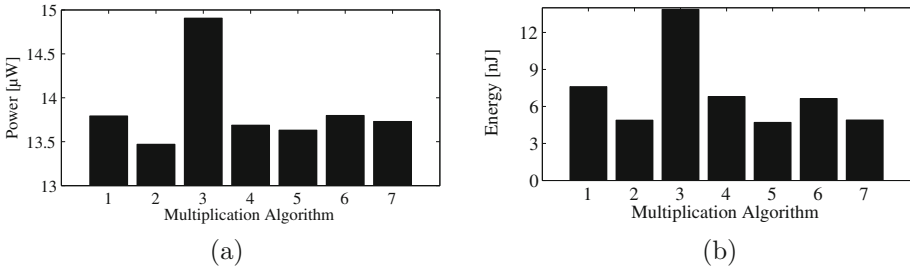
For the power measurements we made use of a LeCroy WaveRunner HRO 66Zi digital sampling oscilloscope. As the MSP-EXP430FR5969 Launchpad Evaluation Kit has been developed to facilitate power measurements, we could easily place a  $2.2\,\Omega$  shunt resistor at the Vdd path of the MSP430FR5969 microcontroller while no stabilizing capacitor was placed between the measurement point and the microcontroller. We powered the Evaluation Kit by an external stable power supply and monitored the current passing through the shunt resistor by means of a LeCroy AP 033 differential probe at a sampling rate of 1 GS/s.

Figure 3(a) shows a sample power trace where the parts dedicated to each multiplication are marked. In Fig. 3(b) we also provide a zoomed view of this trace to highlight several—non-periodic—high peaks which we have observed. We have observed the same peaks (but periodic) for a couple of NOP operations as well. The pattern of these high power consumption peaks are not certainly clear to us, but it seems that they are relevant to FRAM accesses. That is because fetching the instructions from the code memory also needs to access the FRAM.

For 1000 random operand pairs we collected 1000 traces, each of which covers the execution of all 7 multiplications with the same operands. Corresponding to each multiplication, each trace is divided into 7 parts and the voltage observed by the differential probe at each sample point is turned into instantaneous power as  $P = V^2/R$ , where  $R = 2.2\,\Omega$ . Average of instantaneous power values over the period of time corresponding to each multiplication gives us the power consumption of the device for that operation. We also can turn this value to amount of energy the device consumed by  $P \cdot t$ , where  $t$  stands for the duration of the multiplication. Figure 4 depicts the average of power and energy consumption of the microcontroller for each multiplication. Note that since the



**Fig. 3.** A sample power trace measured from MSP-EXP430FR5969 Launchpad Evaluation Kit when running 7 different multiplications



**Fig. 4.** Average of (a) power and (b) energy consumption of different multiplications (the indices for the algorithms fit to the same order shown in Table 1.)

MSP430FR5969 microcontroller on the Evaluation Kit operates by the internal oscillator (8 MHz), the duration of each multiplication was not completely the same for all 1000 measurements due to the small jitter of the oscillator.

As shown by the graphics, 32-bit operand-caching has the lowest power consumption. However, 32-bit Karatsuba consumes less energy as it is the fastest one (see Table 1). As stated above, using 32-bit Karatsuba the debugging functionality of IAR Embedded Workbench IDE reports 6,513,011 clock cycles for the execution of a scalar multiplication on Curve25519 on the board having a

MSP430FR5969. We verified this result measuring the length of the power trace. Based on our practical measurements one full execution of the algorithm takes around 821 ms with operation frequency of 8 MHz. This confirms the cycle count measured with IAR debugging functionality. To measure its power consumption we had to decrease the sampling rate to 200 MS/s due to the length of the trace (825 ms). Based on 100 measurements for random operands, in average the corresponding power consumption and energy consumption is  $14.046\mu\text{W}$  and  $11.623\mu\text{J}$  respectively.

## 6 Conclusion

This paper is the first that presents a full constant-time implementation of Curve25519 on different MSP430 microcontrollers. In order to evaluate and improve the efficiency, we implemented and analyzed different finite-field multiplication techniques and compared them in terms of speed, code size, and power consumption. Amongst all considered multiplication techniques, the subtractive Karatsuba implementation proposed in [14] performs the best. It turned out that 2-level Karatsuba performs better than 1-level Karatsuba in case a  $16 \times 16$ -bit hardware multiplier is available. This is however not the case if the MSP430 has a  $32 \times 32$ -bit hardware multiplier. We further analyzed our implementation with the MSP-EXP430FR5969 Launchpad Evaluation Kit. We presented numbers for the average power and the energy consumption of Curve25519 on this platform. We showed that with an energy consumption of  $11.623\mu\text{J}$  the execution of high-security ECC is feasible on devices operated with battery or harvested power, such as medical implants.

## References

1. Acıgmez, O., Brumley, B.B., Grabher, P.: New results on instruction cache attacks. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 110–124. Springer, Heidelberg (2010). <http://www.iacr.org/archive/ches2010/62250105/62250105.pdf>. 32
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). <http://cr.yp.to/papers.html#curve25519>. 32, 33, 35, 36, 37
3. Bernstein, D.J.: Batch binary edwards. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 317–336. Springer, Heidelberg (2009). <http://cr.yp.to/papers.html#bbe>. 38
4. Bernstein, D.J.: Cryptography in NaCl (2009). <http://cr.yp.to/highspeed/naclcrypto-20090310.pdf>. 35
5. Bernstein, D.J., Schwabe, P.: NEON crypto. In: Prouff, E., Schaumont, P. (eds.) CHES 2012. LNCS, vol. 7428, pp. 320–339. Springer, Heidelberg (2012). <http://cryptosith.org/papers/neoncrypto-20120320.pdf>. 33
6. Bernstein, D.J., van Gastel, B., Janssen, W., Lange, T., Schwabe, P., Smetsters, S.: TweetNaCl: A crypto library in 100 tweets (to appear). Document ID: c74b5bbf605ba02ad8d9e49f04aca9a2. <http://cryptojedi.org/papers/#tweetnacl>. 33

7. Costigan, N., Schwabe, P.: Fast elliptic-curve cryptography on the cell broadband engine. In: Preneel, B. (ed.) AFRICACRYPT 2009. LNCS, vol. 5580, pp. 368–385. Springer, Heidelberg (2009). 33, 36
8. Aranha, D.F., Gouvea, C.P.L.: RELIC is an Efficient Library for Cryptography (2014). <http://code.google.com/p/relic-toolkit/>. Accessed 06 September 2014. 34
9. Gouvea, C.P.L., Lopez, J.: Software implementation of pairing-based cryptography on sensor networks using the MSP430 microcontroller. In: Roy, B., Sendrier, N. (eds.) INDOCRYPT 2009. LNCS, vol. 5922, pp. 248–262. Springer, Heidelberg (2009). <http://conradopl.g.cryptoland.net/files/2010/12/indocrypt09.pdf>. 33, 43
10. Gouvea, C.P.L., Oliveira, L.B., Lopez, J.: Efficient software implementation of public-key cryptography on sensor networks using the MSP430X microcontroller. *J. Crypt. Eng.* **2**(1), 19–29 (2012). <http://conradopl.g.cryptoland.net/files/2010/12/jcen12.pdf>. 42
11. Guajardo, J., Blumel, R., Krieger, U., Paar, C.: Efficient implementation of elliptic curve cryptosystems on the TI MSP430x33x family of microcontrollers. In: Kim, K. (ed.) PKC 2001. LNCS, vol. 1992, pp. 365–382. Springer, Heidelberg (2001). 33
12. Halperin, D., Heydt-Benjamin, T.S., Ransford, B., Clark, S.S., Defend, B., Morgan, W., Fu, K., Kohno, T., Maisel, W.H.: Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In: IEEE Symposium on Security and Privacy - IEEE S&P 2008d, pp. 129–142. IEEE Computer Society (2008). <http://www.secure-medicine.org/public/publications/icd-study.pdf>. 32
13. Hutter, M., Schwabe, P.: NaCl on 8-Bit AVR microcontrollers. In: Youssef, A., Nitaj, A., Hassanien, A.E. (eds.) AFRICACRYPT 2013. LNCS, vol. 7918, pp. 156–172. Springer, Heidelberg (2013). <http://cryptojedi.org/papers/avrnac1-20130220.pdf>. 33
14. Hutter, M., Schwabe, P.: Multiprecision multiplication on AVR revisited (2014). <http://cryptojedi.org/papers/#avrmul>. 37, 39, 45
15. 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://online.tugraz.at/tug\\_online/voe\\_main2.getvolltext?pCurrPk=58138](https://online.tugraz.at/tug_online/voe_main2.getvolltext?pCurrPk=58138). 37, 38
16. T.I. Incorporated: Enabling secure portable medical devices with TI’s MSP430 MCU and wireless technologies (2012). <http://www.ti.com/lit/wp/slay027/slay027.pdf>. 32
17. T.I. Incorporated: MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx family user’s guide (2012). 40
18. T.I. Incorporated: MSP430x2xx family - user’s guide, July 2013. <http://www.ti.com/lit/ug/slau144j/slau144j.pdf>. 34
19. T.I. Incorporated: MSP430F261x datasheet (rev. K) (2014). <http://www.ti.com/lit/ds/symlink/msp430f2618.pdf>. 34
20. Janssen, W.: Curve25519 in 18 tweets. Bachelor’s thesis, Radboud University Nijmegen (2014). [http://www.cs.ru.nl/bachelorscripties/2014/Wesley\\_Janssen\\_\\_\\_4037332\\_\\_\\_Curve25519\\_in\\_18\\_tweets.pdf](http://www.cs.ru.nl/bachelorscripties/2014/Wesley_Janssen___4037332___Curve25519_in_18_tweets.pdf). 33
21. Karatsuba, A., Ofman, Y.: Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, **7**, 595–596 (1963). Translated from *Doklady Akademii Nauk SSSR*, Vol. 145, No. 2, pp. 293–294, July 1962. 37, 39
22. 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, pp. 245–256. IEEE (2008). [discovery.csc.ncsu.edu/pubs/ipsn08-TinyECC-IEEE.pdf](http://discovery.csc.ncsu.edu/pubs/ipsn08-TinyECC-IEEE.pdf). 34

23. C.U. Ltd.: MIRACL cryptographic SDK (2011). <http://www.certivox.com/miracl/> (Accessed 06 September 2014). 34
24. Montgomery, P.L.: Speeding the pollard and Elliptic Curve methods of factorization. *Math. Comput.* **48**(177), 243–264 (1987). 32, 35
25. Pendl, C., Pelnar, M., Hutter, M.: Elliptic curve cryptography on the WISP UHF RFID tag. In: Juels, A., Paar, C. (eds.) *RFIDSec 2011*. LNCS, vol. 7055, pp. 32–47. Springer, Heidelberg (2012). 33
26. Sasdrich, P., Güneysu, T.: Efficient elliptic-curve cryptography using curve25519 on reconfigurable devices. In: Goehringer, D., Santambrogio, M.D., Cardoso, J.M.P., Bertels, K. (eds.) *ARC 2014*. LNCS, vol. 8405, pp. 25–36. Springer, Heidelberg (2014). [https://www.hgi.rub.de/media/sh/veroeffentlichungen/2014/03/25/paper\\_arc14\\_curve25519.pdf](https://www.hgi.rub.de/media/sh/veroeffentlichungen/2014/03/25/paper_arc14_curve25519.pdf). 33
27. Scott, M., Szczechowiak, P.: Optimizing multiprecision multiplication for public key cryptography. *Cryptology ePrint Archive*, Report 2007/299 (2007). <http://eprint.iacr.org/2007/299/>. 33
28. I. Systems: IAR C/C++ Compiler reference guide for texas instruments’ msp430 microcontroller family (2011). 32, 34
29. Szczechowiak, P., Kargl, A., Scott, M., Collier, M.: On the application of pairing based cryptography to wireless sensor networks. In: Basin, D.A., Capkun, S., Lee, W. (eds.) *Proceedings of the Second ACM Conference on Wireless Network Security - WiSec 2009*, pp. 1–12. ACM (2009). 33
30. 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.) *EWSN 2008*. LNCS, vol. 4913, pp. 305–320. Springer, Heidelberg (2008). <http://www.ic.unicamp.br/~leob/publications/ewsn/NanoECC.pdf>. 34
31. Tromer, E., Osvik, D.A., Shamir, A.: Efficient cache attacks on AES, and counter-measures. *J. Cryptol.* **23**(1), 37–71 (2010). <http://www.tau.ac.il/tromer/papers/cache-joc-20090619.pdf>. 32
32. Wenger, E., Unterluggauer, T., Werner, M.: 8/16/32 shades of elliptic curve cryptography on embedded processors. In: Paul, G., Vaudenay, S. (eds.) *INDOCRYPT 2013*. LNCS, vol. 8250, pp. 244–261. Springer, Heidelberg (2013). 33, 43
33. Wenger, E., Werner, M.: Evaluating 16-bit processors for elliptic curve cryptography. In: Prouff, E. (ed.) *CARDIS 2011*. LNCS, vol. 7079, pp. 166–181. Springer, Heidelberg (2011). 33, 43

Progress in Cryptology - LATINCRYPT 2014  
Third International Conference on Cryptology and  
Information Security in Latin America Florianópolis,  
Brazil, September 17-19, 2014 Revised Selected Papers  
Aranha, D.F.; Menezes, A. (Eds.)  
2015, XII, 387 p. 64 illus., Softcover  
ISBN: 978-3-319-16294-2