Energy-Efficient Elliptic Curve Cryptography for MSP430-Based Wireless Sensor Nodes

Zhe Liu[1], Johann Großschädl[2], Lin Li[2,3], and Qiuliang Xu[3]

[1] University of Waterloo, Canada
z446liu@uwaterloo.ca
[2] University of Luxembourg, Luxembourg
johann.groszschaedl@uni.lu
[3] Shandong University, China
vivililin@gmail.com, xql@sdu.edu.cn

Abstract. The Internet is rapidly evolving from a network of personal computers and servers to a network of smart objects ("things") able to communicate with each other and with central resources. This evolution has created a demand for lightweight implementations of cryptographic algorithms suitable for resource-constrained devices such as RFID tags and wireless sensor nodes. In this paper we describe a highly optimized software implementation of Elliptic Curve Cryptography (ECC) for the MSP430 series of ultra-low-power 16-bit microcontrollers. Our software is scalable in the sense that it supports prime fields and elliptic curves of different order without recompilation, which allows for flexible trade-offs between execution time (i.e. energy consumption) and security. The low-level modular arithmetic is optimized for pseudo-Mersenne primes of the form $p = 2^n - c$ where n is a multiple of 16 minus 1 and c fits in a 16-bit register. All prime-field arithmetic functions are parameterized with respect to the length of operands (i.e. the number of 16-bit words they consist of) and written in Assembly language, whereby we avoided conditional jumps and branches that could leak information about the secret key. Our ECC implementation can perform scalar multiplication on two types of elliptic curves, namely Montgomery curves and twisted Edwards curves. A full scalar multiplication using a Montgomery curve over a 159-bit field requires about $3.86 \cdot 10^6$ clock cycles when executed on an MSP430F1611 microcontroller.

Keywords: Internet of things, lightweight cryptography, Montgomery curve, twisted Edwards curve, multi-precision arithmetic.

1 Introduction

More and more non-traditional computing devices, ranging from various kinds of sensors and actuators over consumer electronics and household appliances to smart vehicles and related road-side infrastructure, get equipped with wireless transceivers, which allows them to communicate with each other or connect to the Internet. According to a whitepaper by Cisco, the number of smart objects (or “things”) connected to the Internet started to exceed the number of people with Internet access at some time between 2008 and 2009 [7]. Consequently, the so-called *Internet of Things (IoT)* has become reality some seven years ago. In the same whitepaper, Cisco estimates that the IoT will encompass no less than 50 billion “things” by the year 2020, which corresponds to some 6.58 connected devices per person. However, since this is an average figure, it can be assumed that, in the developed world, every person will soon be surrounded by dozens of smart devices capable to interact with each other in an entirely autonomous fashion. The IoT is expected to have a profound impact on every sector of the economy, ranging from agriculture to high tech, and touch our daily lives to a much larger extent than the Internet did in the past 20 years [16]. At the same time, it is also clear that 50 billion smart devices connected to the Internet pose unprecedented challenges to the security and privacy of their owners or users.

Similar to the “traditional” Internet (i.e. the Internet connecting commodity computers and servers), public-key cryptography plays a crucial role in the security arena of the IoT. In the recent past, a number of lightweight variants of common security protocols have been proposed (e.g. Datagram TLS [29] and HIP Diet Exchange [25]), taking the very specific requirements and constraints of the IoT into account [13]. These protocols support the use of Elliptic Curve Cryptography (ECC) [12] as a less-costly alternative to RSA [30] for such tasks as authentication and key establishment. However, even though ECC features a much better security-per-bit ratio than RSA, it is still computation intensive and, therefore, poses a massive burden for resource-constrained IoT devices. In fact, a large number of the smart objects that populate the IoT only feature an 8 or 16-bit processor clocked with a frequency of a few MHz [31]. A well-known example for a 16-bit platform targeted towards IoT applications is the MSP430 family of ultra-low power microcontrollers from Texas Instruments [5]. Besides modest processing power, IoT devices often possess only a few kB of RAM and (at most) a few 100 kB of Flash memory. However, for battery-powered devices such as wireless sensor nodes, *energy* is by far the most precious resource. Once deployed in the field, a sensor node is expected to run several months, or even years, without any maintenance and replacement, which means it must survive for long periods of time on a single battery charge.

Past research on improving the energy efficiency of ECC software focussed primarily on reducing the execution time of *scalar multiplication*, which is the main arithmetic operation of virtually all elliptic curve cryptosystems [12]. The energy consumption of (cryptographic) software on an embedded processor is closely tied to its execution time in the sense that a faster execution of a given algorithm normally translates to savings in energy [33, Sect. 6.3]. Possibilities for reducing the execution time of a scalar multiplication exist at both the field and the curve arithmetic layer, respectively. Related to the former are various approaches to speed up multiple-precision arithmetic on the MSP430 platform [9, 22, 27, 32, 37]. Recently, Düll et al [6] reported impressive new speed records for multiplication (and squaring) modulo a 255-bit prime, which they achieved through an elaborate implementation of Karatsuba’s technique [17] combined

Table 1. Classes of constrained devices (source: RFC 7228 [4])

Name	Data size (e.g. RAM)	Code size (e.g. Flash)
Class 0, C0	$\ll$ 10 kB	$\ll$ 100 kB
Class 1, C1	$\sim$ 10 kB	$\sim$ 100 kB
Class 2, C2	50 kB	250 kB

with sophisticated Assembly optimizations. In a second line of research, the impact of alternative curve models, such as Montgomery [24] or twisted Edwards curves [3], has been studied for both high [6] and low to medium security levels [20]. The Montgomery model currently holds the speed record for variable-base scalar multiplication on MSP430 processors, while the twisted Edwards model achieves record-setting performance for fixed-base scalar multiplication. Due to their excellent performance characteristics, Montgomery and twisted Edwards curves also occupy the top spots in terms of energy efficiency.

Besides minimizing execution time, there exists a second avenue to reduce the overall energy consumption of ECC operations performed in an IoT device (e.g. a wireless sensor node), namely to deploy *energy-scalable ECC software*. In cryptographic engineering, the term scalability generally refers to the ability to process operands (including keys) of any length without introducing the need to re-design or re-implement a cryptosystem. More concretely, ECC software is said to be energy-scalable if it can perform scalar multiplication in elliptic-curve groups of varying order (i.e. varying cryptographic strength) without having to re-write and/or re-compile the source code [21]. Energy-scalable ECC software allows for flexible and dynamic (i.e. run-time adaptable) trade-offs between security and energy consumption, whereby the execution time (and thus also the energy cost) of a scalar multiplication increases with the cube of the group size in bits. The real-world benefit of energy-scalable cryptographic software in the context of the IoT is probably best explained by taking a Wireless Sensor Network (WSN) [28] as example. A WSN may utilize ECC for such tasks as access control [38], key exchange [20, 21], or broadcast/multicast authentication [8], to list a few. While all these tasks are security critical, their actual requirements with respect to the hardness of the Elliptic Curve Discrete Logarithm Problem (ECDLP) differ greatly, as we will explain in the following section. A scalable ECC implementation allows one to use smaller groups for less security-critical tasks, thereby saving precious energy. In contrast, ECC software that supports just a single curve falls short in terms of either security or energy efficiency.

Besides energy, also RAM and non-volatile memory (i.e. ROM or Flash) are scarce resources on MSP430-based sensor nodes and other IoT devices. This is little surprising since MSP430 microcontrollers have a common address space of 16 bits for RAM and Flash, which means the addressable memory space is only 64 kB (i.e. 2^{16} bytes) [36, Sect. 1.4]. The IETF distinguishes among three classes of constrained devices on basis of their memory capacity as specified in Table 1. Class 0 devices are so restricted in memory and processing capabilities that, according to [4], “most likely they will not have the resources required to communicate directly with the Internet in a secure manner." A typical example of a Class 0 device is the WISP 4.1DL, a software-programmable passive RFID tag equipped with an MSP430F2132 microcontroller that features 512 B RAM and 8 kB Flash memory [34, p. 121]. Most wireless sensor nodes fall into Class 1; for example, the TelosB [23] contains an MSP430F1611 microcontroller and provides 10 kB RAM as well as 48 kB Flash. Even though the TelosB is able to store six times more program code than the WISP RFID tag, it must be taken into account that a sensor-node operating system alone may, depending on the configuration, use up half of the TelosB's Flash memory [11]. In addition, since these operating systems do not support any security services other than simple link-layer encryption, application developers have to include auxiliary libraries for security protocols (e.g. Datagram TLS [29]) and the required cryptographic primitives, which significantly increases the total code size. Therefore, it can be expected that, in real-world application scenarios, only a small fraction of the TelosB's Flash capacity of 48 kB will actually be available for ECC.

In this paper, we introduce an energy-scalable ECC implementation for the 16-bit MSP430 platform that supports both Montgomery and twisted Edwards curves over pseudo-Mersenne prime fields. More precisely, our ECC software is able to carry out variable-base scalar multiplication on Montgomery curves and fixed-base scalar multiplication on twisted Edwards curves. The low-level field arithmetic is optimized for primes of the form $p = 2^n - c$ where n is a multiple of 16 minus 1 (e.g. $n = 159, 191, 223, 255$) and c has a length of at most 15 bits so that it fits into a single register of an MSP430 processor. All functions of the $\mathbb{F}_p$-arithmetic library are parameterized by a "length" parameter, which means they take an argument that specifies the number of 16-bit words the operands (resp. result) consist of. In this way, one and the same arithmetic function can be used for pseudo-Mersenne prime fields of different order (e.g. ranging from 159 to 255 bits) without having to re-compile the source code. Contrary to the majority of previous work (e.g. [6, 14, 20]), we aimed for a compromise between performance and code size rather than optimizing solely for speed. In particular, we strived for an implementation that is small enough to fit into the code space of Class 0 and Class 1 devices as listed in Table 1 [4]. Achieving a good trade-off between speed and size is a highly challenging task since the common measures to improve performance (e.g. loop unrolling, storage of pre-computed curve points) have a negative impact on ROM/Flash consumption.

In order to reach high performance, we implemented all low-level operations supported by our $\mathbb{F}_p$-arithmetic library in Assembly language, but we refrained from adopting code-size increasing optimizations such as loop unrolling. When classifying ECC software along an axis between speed and size, our implementation is close to the far end towards size because we sacrificed speed for small code size, while most previous implementations sacrificed code size to achieve high speed. The whole $\mathbb{F}_p$-arithmetic library we describe in this paper occupies just some 2.3 kB in Flash memory, which makes our implementation one of the most compact ever reported in the literature. To put this into perspective, the fully-unrolled implementation of multiplication in a 255-bit prime field recently introduced by Düll et al [6] has a size of roughly 3.6 kB when compiled for an MSP430 processor, i.e. their multiplication function alone consumes 56% more code space than our complete $\mathbb{F}_p$-arithmetic library. We also made an effort to protect the library against timing attacks [18] by implementing all arithmetic operations, except inversion, in a "regular" fashion so that they always execute exactly the same sequence of instructions. Hence, the execution time depends solely on the length of the involved operands (i.e. the number of 16-bit words they consist of), but not on their actual values. We implemented the inversion via the extended Euclidean algorithm, which has operand-dependent execution time. However, as we will show in this paper, timings attacks on the inversion can be can be effectively thwarted through multiplicative masking. We provide a detailed execution time analysis of various of field-arithmetic operations and scalar multiplication algorithms on an MSP430F1611 processor [36] for Montgomery and twisted Edwards curves over 159, 191, 223 and 255-bit fields.

2 WSN Security: Confidentiality Versus Integrity

A WSN can be defined as a wireless network of autonomous sensor nodes (also called *motes* [23]) that are spatially distributed in a certain area of interest to cooperatively monitor a certain physical phenomenon or condition like temperature, humidity, light, smoke, vibrations, etc. [28]. In recent years, WSNs have found widespread adoption in a multitude of application domains ranging from medical monitoring over home automation and traffic control to environmental surveillance. A typical sensor node is an inexpensive, battery-operated device equipped with an 8 or 16-bit microcontroller and has limited memory resources [23]. The way in which the nodes collect, process and transfer sensor readings depends on various factors such as the size, topology, and organization of the WSN. In the simplest case, each node sends its sensor readings directly to the Base Station (BS) or some other data collector for post-processing and decision making. However, this approach wastes precious energy for data transfers since the senor readings of neighboring nodes are typically very similar and, hence, a lot of redundant data gets sent to the BS. In a large-scale WSN, the nodes are often organized in clusters, in which some pre-processing of sensor data, called data aggregation [28], is performed to filter out redundant or correlated data with the goal of reducing network traffic. Data transmission in a large WSN is usually multi-hop in the sense that the nodes cooperatively receive and forward (i.e. relay) packets from the source towards the destination.

WSNs differ greatly from traditional networks regarding threat models and assumptions about the attacker's capabilities [28]. Traditional research in network security was always conducted under the assumption that the endpoints of a communication channel are secure. On the other hand, this is not the case in WSN security research since the attacker is assumed to have physical access to the sensor nodes, which allows him to manipulate or even capture a certain number of nodes. After capturing a node, the attacker will be able to obtain all data stored on it (including cryptographic keys) because wireless sensor nodes are not tamper resistant due to cost reasons. The attacker may even manage to re-program a captured node and integrate it back into the WSN, where it then can conduct all kinds of malicious activities. In contrast to ordinary nodes, the BS possesses plenty of resources and is out of reach from physical attacks.

There exists a massive body of literature with proposals on how ECC could contribute to improve the security of WSNs [1, 21, 38]. The "big picture" of an ECC-enabled WSN can be sketched as follows. Prior to deployment, each node gets equipped with a key pair suitable for ECDH key agreement, whereby the public key along with the node's network address is signed by a trusted third party, which is called Central Authority (CA) in [1]. In practice, this signature can have the form of a lightweight certificate that contains an expiration date and other relevant information. Besides the key pair and certificate, each node is also loaded with the public key corresponding to the trusted party's private signing key, which allows the node to check the validity of a certificate. During the initial configuration (i.e. bootstrapping) of the WSN, each node exchanges certificates with its neighbors within communication distance and then verifies the validity of all received certificates with help of the public key of the trusted party. The nodes establish secret keys with all neighbors that are in possession of a valid certificate using static ECDH key exchange (ephemeral ECDH would also be possible, but is slightly more complicated). Now, neighboring nodes can form a cluster, select a cluster head, and carry out various other initialization activities. From time to time, e.g. when a node runs out of battery, a new node needs to be integrated into the WSN. The cluster head verifies the certificate of the new node and sends the node's network address to the BS so that it can update routing tables and perform other management tasks.

In the ECC-based security architecture for WSNs sketched above, ECDH is used for key agreement between neighboring nodes and ECDSA for signatures (e.g. certificates). These two cryptosystems have greatly different requirements regarding their "strength" (i.e. the hardness of the ECDLP), which also reflects the damage an adversary can cause when breaking them. For example, when an attacker manages to obtain the private ECDH key of a node, he may be able to decrypt the messages sent between this node and its neighbors. For many real-world applications, such a confidentiality breach in a small part of the WSN is a non-critical issue (as mentioned in e.g. [35]) since, despite this data leak, the proper functioning of the WSN is not impacted. On the other hand, when the attacker manages to get hold of the trusted party's signing key, he is able forge signatures and certificates, which will allow him to inject malicious nodes into the WSN. Such malicious nodes can, for example, manipulate messages on the way from source to destination or completely drop packets. The consequences of such an integrity breach can be disastrous and may, in the words case, make the whole WSN useless [28]. Therefore, long-term signature keys used for node authentication deserve a higher level of security than e.g. ECDH keys used to establish shared secrets between nodes. It is, of course, possible to do both the node authentication and key exchange with a high-security elliptic curve, but in such a setting the key exchange is "over-secured," which wastes energy.

3 Prime-Field Arithmetic

The performance of virtually any ECC implementation depends heavily on the execution time of certain arithmetic operations (in particular multiplication) in the underlying finite field. When implementing ECC in software, it is common practice to use curves defined over some special prime fields that facilitate the modular reduction operation [12]. A well-known example for primes with good arithmetic properties are the so-called *pseudo-Mersenne primes* [12], which are primes of the form of $p = 2^n \pm c$ where c is relatively small (to fit into a single register of the target processor) and n is chosen such that p meets the desired bitlength. When taking advantage of the congruence relation $2^n \equiv c \bmod p$, the reduction of a $2n$-bit integer modulo $p = 2^n - c$ has linear complexity, i.e. the execution time of the reduction operation increases linearly with n [12].

The NIST and numerous other standards bodies recommend elliptic curves over prime fields whose bitlengths are multiples of 32, e.g. 192, 224, or 256 bits [12]. However, as demonstrated by Bernstein in [2], it can be more beneficial to use primes that are slightly shorter than the standard bitlengths (e.g. 255 bits instead of 256), especially if one aims for both high speed and a regular execution profile to counter timing attacks. Having one bit of "slack space" permits some special software optimization techniques that are not applicable when the bitlength is exactly a multiple of the processor's word size. The elliptic curves we used to benchmark our implementation follow this approach since they are defined over fields given by primes of the form $p = 2^n - c$ where n is a multiple of 32 minus 1 (e.g. $n = 159$ or 191) and c is up to eight bits long. However, the parameterized $\mathbb{F}_p$-arithmetic library for MSP430 processors we describe in this section is flexible enough to support any n that is a multiple of 16 minus 1 and any c that is in the range of $[1, 2^{15} - 1]$.

We use the following notation to describe the arithmetic functions: n is the bitlength of the prime $p = 2^n - c$; this implies that any element $a \in \mathbb{F}_p$ is also n bits long. When working on a w-bit processor, a field element can be stored in an array of $m = \lceil n/w \rceil$ words, each consisting of w bits. In our case, $w = 16$ since the target platform is an MSP430 processor. We use lowercase letters to denote field elements and indexed lowercase letters to refer to individual words of a field element. Consequently, $a \in \mathbb{F}_p$ can be written as an array of the form $(a_{m-1}, \ldots, a_1, a_0)$ where a_{m-1} is the Most Significant Word (MSW) and a_0 the Least Significant Word (LSW). All arithmetic functions of our library are able to process incompletely-reduced operands, which means that an operand does not necessarily need to be the least non-negative residue modulo p, but has to fit into m words and can, therefore, be up to $n + 1$ bits long. Also the results produced by our functions for $\mathbb{F}_p$-arithmetic may not be fully reduced.

3.1 Modular Addition and Subtraction

The straightforward way to perform a modular addition $r = a + b \bmod p$ is to first calculate the sum $s = a + b$ and then subtract a multiple of p from it to obtain a final result that fits into m words. Since each of the operands can be

$n+1$ bits long, the subtrahend is either 0, p, $2p$, or $3p$ [20]. However, instead of subtracting a multiple of p, it is, in general, more efficient to add a multiple of c, which is possible since $2^n \equiv c \bmod p$. An MSP430 implementation of this approach for modular addition with fully-unrolled loops and regular execution profile is described in [20, Sect. 3.1]. The main drawback of such an "add-then-subtract" technique is that it consists of two loops, each introducing overhead if they are not unrolled. Due to the simplicity of these loops, the loop overhead (i.e. updating of a loop counter, checking of a loop-termination condition, and jumping back to the start) can significantly impact the overall performance.

Since we aim for small code size (and, therefore, avoid loop unrolling), it is very important to minimize the loop overhead. An obvious way to achieve this is to employ a modular addition technique that requires only a single loop, like the one described by Düll et al in [6, Sect. 4.4]. Our implementation is based on their approach and performs an addition in $\mathbb{F}_p$ as follows. First, we add the MSWs of a and b, i.e. we compute the sum $s = a_{m-1} + b_{m-1}$, which can be up to 17 bits long when the operands are incompletely reduced. This sum is then split up into a lower part s_L consisting of the 15 least-significant bits, and an upper part s_H with the remaining two bits. We temporarily store s_L in a register and multiply s_H by c. Thereafter, the $m-1$ remaining words of the two operands are added (with carry propagation), starting with the LSWs a_0 and b_0. The main difference to a "conventional" multi-precision addition is that the product of s_H and c is added to the two LSWs and, therefore, the carry to be propagated to the next-higher word can be 0, 1, or 2. Finally, the carry from the last addition (i.e. the addition of the words a_{m-2} and b_{m-2}) is propagated into s_L, which is then at most 16 bits long. The final result has a length of no more than $n+1$ bits (i.e. fits into m words), but may be not fully reduced.

The most basic way to perform a modular subtraction $r = a - b \bmod p$ consists of two simple steps: the computation of the difference $d = a - b$, followed by an addition of p (or a multiple thereof) to get a non-negative result. In the most extreme case, namely when $a = 0$ and b has the maximum possible value of $2^{n+1} - 1$, it is necessary to add $3p$. A constant-time implementation of this modular subtraction technique with unrolled loops is described by Düll et al in [6, Sect. 4.4]. However, when implemented with "rolled" loops to minimize the code size, this approach suffers from a high overhead since (at least) two loops need to be executed; one for the actual subtraction and the other(s) to obtain a non-negative result. In order to minimize the loop overhead, we perform the modular subtraction by computing $r = 4p + a - b \bmod p$ since this operation can be implemented with one single loop, similar to the modular addition. The addition of $4p$, which ensures that the final result is positive, does not cause much overhead when taking into account that all w-bit words of $4p$, except the two LSWs and the MSW, are identical and can be kept in a register.

3.2 Multiplication and Squaring

Some MSP430 models, including our target processor (the MSP430F1611), are equipped with a hardware multiplier that is capable to perform multiplications

Algorithm 1. Multiple-precision multiplication (product-scanning method)

Input: Two m-word operands $a = (a_{m-1}, \ldots, a_1, a_0)$ and $b = (b_{m-1}, \ldots, b_1, b_0)$
Output: $2m$-word product $z = a \times b = (z_{2m-1}, \ldots, z_1, z_0)$

```
$s \leftarrow a_0 \times b_0$
$z_0 \leftarrow s \bmod 2^w$ ; $s \leftarrow s/2^w$
for $i$ from 1 by 1 to $m-1$ do
    $j \leftarrow 0$ ; $k \leftarrow i$
    while $k \geq 0$ do
        $s \leftarrow s + a_j \times b_k$
        $j \leftarrow j+1$ ; $k \leftarrow k-1$
    end while
    $z_i \leftarrow s \bmod 2^w$ ; $s \leftarrow s/2^w$
end for
for $i$ from $m$ by 1 to $2m-3$ do
    $j \leftarrow i-(m-1)$ ; $k \leftarrow m-1$
    while $j \leq m-1$ do
        $s \leftarrow s + a_j \times b_k$
        $j \leftarrow j+1$ ; $k \leftarrow k-1$
    end while
    $z_i \leftarrow s \bmod 2^w$ ; $s \leftarrow s/2^w$
end for
$s \leftarrow s + a_{m-1} \times b_{m-1}$
$z_{2m-2} \leftarrow s \bmod 2^w$
$z_{2m-1} \leftarrow s/2^w$
return $(z_{2m-1}, \ldots, z_1, z_0)$
```

and Multiply-ACcumulate (MAC) operations with 16-bit integers. This multiplier is not tightly coupled to the processor core, but attached to it in the form of a memory-mapped peripheral. The MSP430 instruction set does not include dedicated multiply or MAC instructions; instead, the multiplier is accessed via a set of eight peripheral registers that are visible in the address space and can be loaded and read using the `mov` instruction [36]. There are four registers (and associated memory addresses) for the first operand, called `MPY`, `MPYS`, `MAC`, and `MACS`, as well as one register and address for the second operand, referred to as `OP2`. The type of operation (i.e. multiplication or MAC, signed or unsigned) is selected by the address the first operand is written to. For example, to perform an unsigned multiplication, one has to write the first operand to `MPY`. The execution of the selected operation starts immediately and automatically after the second operand is written to `OP2`. There are three result registers: `RESLO` holds the lower 16 bits of the result, `RESHI` the higher 16 bits, while `SUMEXT` contains the carry of the accumulate operation if an unsigned MAC is performed.

From an algorithmic point of view, there are two basic techniques to implement a multiple-precision multiplication, namely the operand-scanning method and the product-scanning method (see [12, 19] for details). The availability of a hardware-supported MAC operation clearly indicates that the latter technique may reach better performance on an MSP430F1611 device. Algorithm 1 shows our implementation of the product-scanning approach, which is very similar to that in [12, Algorithm 2.10]. It consists of two nested loops; the first computes the lower half of the product z (i.e. the words z_1 to z_{m-1}), whereas the second nested loop contributes the higher words (i.e. z_m to z_{2m-3}). Both inner loops perform MAC operations; in each iteration, two w-bit words are multiplied and the $2w$-bit product is added to a cumulative sum s. In general, when adding up several such $2w$-bit products, the length of the sum s can exceed $2w$ bits. The index j is incremented in the inner loop, while k is decremented, which means the words of operand a are loaded in ascending order and those of operand b in descending order. An operation of the form $z_i \leftarrow s \bmod 2^w$ as in line 9 assigns the w least significant bits of s to the word z_i, whereas $s \leftarrow s/2^w$ represents a w-bit right-shift of s. Note that the computation of $a_0 \times b_0$ is "peeled off" from the first nested loop since it is not a MAC operation but just a straightforward multiplication. We also compute the last $2w$-bit product, $a_{m-1} \times b_{m-1}$, outside the second nested loop because it is not necessary to shift s after the addition of this product; instead, we can directly write s to z_{2m-2} and z_{2m-1}.

Listing 1. First inner loop of the product-scanning method in Assembly language

```
1  INNLOOP1:
2      MOV  @APTR+, &MAC
3      MOV  @BPTR, &OP2
4      SUB  #2, BPTR
5      ADD  @EXTPTR, CARRY
6      CMP  INNSTOP, BPTR
7      JGE  INNLOOP1
```

Listing 1 shows our Assembly implementation of the first inner loop of the product-scanning method (line 5 to 8 of Algorithm 1). `APTR` and `BPTR` are two registers that contain pointers (i.e. addresses) through which the 16-bit words of operand a and b are accessed. The first `MOV` instruction writes a word of a to `MAC`, thereby configuring the multiplier to execute a MAC operation. Then, the second `MOV` instruction writes a word of b to `OP2`, which immediately starts the execution. A MAC operation consists of the multiplication of the 16-bit words written to `MAC` and `OP2`, followed by the accumulation of the 32-bit product to the content of the `RESHI|RESLO` register pair, whereby the resulting carry bit is placed in `SUMEXT`. In line 5 of Listing 1, the carry bit gets added to a general-purpose register named `CARRY` using the indirect addressing mode to read from `SUMEXT` (see [36, Sect. 7.2.4] for further explanations). MSP430 processors have an autoincrement addressing mode, with which we update `APTR`, but there is no autodecrement mode. Therefore, we manually decrement `BPTR` with help of the `SUB` instruction in line 4. The register `INNSTOP` holds the address of b_0 (i.e. the LSW of operand b) and, consequently, the loop iterates as long as the address in `BPTR` is greater or equal to `INNSTOP`. Each iteration takes 16 clock cycles on an MSP430F1611 processor, to which the loop overhead (i.e. the comparison in line 6 and the jump instruction in line 7) contributes three cycles.

Optimized Squaring. Squaring is a special case of multiplication that allows for dedicated optimizations due to the equality of the two operands [12]. When an ordinary multiplication algorithm, such as the product-scanning method, is used for squaring (by setting $b = a$), then all $2w$-bit word-products of the form $a_j \times a_k$ with $j \neq k$ are computed twice because $a_j \times a_k = a_k \times a_j$ [19]. Only the m word-products $a_i \times a_i$, which lie in the "main diagonal" and are themselves squares, are generated and processed only once. Dedicated squaring algorithms intend to avoid such overheads by computing each word-product only once and then doubling it (e.g. trough a left-shift) if needed. When implemented in this way, the squaring of an integer consisting of m words requires the computation of $(m^2 + m)/2$ word-products, which is just about half of the m word-products that have to be formed in the course of an ordinary multiplication. However, in practice, the saving in execution time is often significantly below 50%.

Our implementation of the squaring function in Assembly language follows closely Algorithm 2 in [19]. This algorithm consists of two nested loops, plus a third one, which is a simple ("un-nested") loop. The two nested loops compute the word-products to be doubled and are similar to those of the multiplication in Algorithm 1, except that the termination conditions for the inner loops are different since they are iterated fewer times. Both inner loops consist of just six Assembly instructions (similar to Listing 1), and each iteration takes 16 cycles on our MSP430F1611 device. In the third loop, which is iterated m times, the intermediate result obtained so far is doubled and the m word-products of the form $a_i \times a_i$ (which are actually word-squares) are added. Each iteration of the third loop requires 42 clock cycles on an MSP430F1611 processor.

Modular Reduction. As mentioned in the beginning of this section, our $\mathbb{F}_p$-arithmetic library is scalable and optimized for pseudo-Mersenne primes of the form $p = 2^n - c$, where n is a multiple of 16 minus 1 and c can be up to 15 bits long. The functions of the arithmetic library can process incompletely-reduced operands, which means the operands can exceed their nominal bitlength by one bit and have a length of $n + 1$ bits. Consequently, the result of a multiplication or squaring is up to $2n + 2$ bits long and fits into $2m$ words. A straightforward approach for reducing a $2m$-word product z modulo $p = 2^n - c$ is to split z up into a lower part z_L and a higher part z_H such that $z = z_H \cdot 2^n + z_L$, followed by a substitution of 2^n by c, which is possible since $2^n \equiv c \bmod p$. However, in our case, this method entails some overhead because n is not a multiple of the word size w and, thus, shift operations are necessary to extract z_H from z.

To avoid bit-level shifts, we perform the reduction operation in two steps as described in [6, 20]. In the first step, the $2m$-word product z is reduced modulo $2p$ into an intermediate result t consisting of $m + 1$ words, which is then in the second step further reduced modulo p to yield a final result with m words. The first step requires to partition z into a lower part z_L consisting of the m LSWs (i.e. the $n + 1$ least significant bits) of z and a higher part z_H with the remaining m words. However, this partitioning does not require any shift operations since it is done at the word-size boundary. Then, we compute the intermediate result $t = z_H \cdot 2c + z_L$ using the operand-scanning technique [12], whereby the 16-bit quantity $2c$ needs to be written to MPY only once and can then be used for all m multiplications [36]. This also explains why the length of c is limited to at most 15 bits as otherwise $2c$ would not fit in a register. Note that, when z is a product of two $(n+1)$-bit integers, i.e. when $z \leq (2^{n+1}-1)^2$, and c has a length of $w-1$ bits, then t is at most $n+w+1$ bits long and can be stored in $m+1$ words. In the second step of the reduction operation, t is split up into a lower part t_L with exactly n bits and a higher part t_H with $w+1$ bits. The final result $r = t_H \cdot c + t_L$ is then obtained by multiplying t_H by c, adding the $2w$-bit product to the two LSWs of t_L, and propagating the carry bit up to the MSW. Even though r may not be fully reduced, it fits into m words.

3.3 Inversion

Since inversion in $\mathbb{F}_p$ is an extremely costly arithmetic operation, it is common practice in ECC software to use projective coordinates for scalar multiplication so that only a single inversion has to be carried out to convert the result from projective to affine coordinates [12]. However, this final inversion is a potential source of side-channel leakage as it can reveal information about the projective representation of the point obtained as result, which, in turn, may allow an attacker to learn a few bits of the secret scalar [26]. In order to prevent this kind of attack, the inversion has to be implemented in such a way that its execution time is either constant (i.e. operand independent) or appears random [18]. The most common way to achieve the former is to execute the inversion through an exponentiation according to Fermat's little theorem, i.e. $a^{-1} = a^{p-2} \bmod p$. In most previous papers describing timing-attack-resistant ECC software, such as [6, 20], this exponentiation was implemented using addition chains.

While Fermat-based inversion allows one to achieve constant execution time in a relatively straightforward way, it is significantly slower than the Extended Euclidean Algorithm (EEA) [12]. However, the EEA has an irregular execution profile and, consequently, operand-dependent execution time. Nonetheless, it is possible to thwart timing attacks against EEA-based inversion by employing a simple multiplicative masking method. Let Z be the z-coordinate of a point in projective coordinates and let u be a random element of $\mathbb{F}_p$ that is unknown to the attacker. Instead of inverting Z directly, we first multiply Z by u, then invert the product Zu using the EEA to obtain $(Zu)^{-1}$, and finally multiply the inversion result $(Zu)^{-1}$ by u to get Z^{-1}. In this way, the execution time of the inversion depends on both Z and u, but since the attacker does not know u, he is not able to draw any conclusions about the actual value of Z. Note that, in our ECC software, u is "hard-coded" and can not be changed, which requires us to take care that an attacker can not reveal u by exploiting variations in the execution time. This is especially important in ECDH key exchange where an attacker can use a fake public key, e.g. a point of low order, which may enable him to "guess" Z and use this information to get u. We thwart such attempts by insisting secret scalars to be a multiple of the curve's co-factor, as in [2], so that the resulting Z coordinate is 0 and the inversion is not executed.

4 Point Arithmetic and Scalar Multiplication

In this paper, we consider two special families of elliptic curves, namely Montgomery [24] and twisted Edwards curves [3]. The former achieve unprecedented efficiency in variable-base scalar multiplication, such as performed in static and ephemeral ECDH key exchange, whereas the latter excels in the other two use cases, namely fixed-base scalar multiplication (needed in e.g. ECDSA signature generation and ephemeral ECDH key exchange) and double-base scalar multiplication (performed in e.g. the verification of an ECDSA signature).

4.1 Montgomery Curve

The Montgomery model of an elliptic curve was originally introduced 20 years ago to speed up algorithms for integer factorization [24]. Formally, a so-called Montgomery curve over $\mathbb{F}_p$ can be defined by an equation of the form

$$E_M: \ By^2 = x^3 + Ax^2 + x \tag{1}$$

where $A, B \in \mathbb{F}_p$ and $(A^2 - 4)B \neq 0$ [24]. Montgomery curves feature a unique addition law that is "special" in two aspects. First, the addition law describes a so-called differential addition, which means it allows one to compute the sum $P_1 + P_2$ of two points P_1, P_2 whose difference $P_1 - P_2$ is known. Second, when using projective coordinates, both a point addition and point doubling can be performed using X and Z coordinates only, i.e. the Y coordinate is not needed for the computation. The usual approach to implement scalar multiplication on a Montgomery curve is to use the Montgomery ladder [24], a simple algorithm that executes both a (differential) addition and a doubling for each bit of the scalar, independent of its actual value. Therefore, the Montgomery ladder has a regular execution profile, which helps to thwart side-channel attacks.

We implemented the differential point addition and point doubling on basis of the formulae given in [24]. Consequently, the point addition consists of three multiplications, four squarings, three additions, and the same number of subtractions in $\mathbb{F}_p$. A doubling, on the other hand, takes two multiplications, two squarings, two additions, two subtractions, as well as a multiplication by the constant $(A + 2)/4$, which is a relatively cheap operation if the parameter A is chosen properly [2]. There are two implementation options for the ladder; one is the standard approach with separate addition and doubling functions, while the other combines both into a so-called "ladder step" [6]. The variant with the ladder step requires besides the normal $\mathbb{F}_p$-arithmetic operations also a special function for conditional swapping of two field elements, but has the advantage of a highly regular memory access pattern (i.e. the addresses used to load and store intermediate results do not depend on secret information). We decided to use the standard approach since MSP430 devices have no cache (and, thus, do not leak timing information through secretly-indexed loads) and since we found it a little faster than the ladder-step variant when taking into account that the three least-significant bits of a scalar are 0 and require only doublings.

4.2 Twisted Edwards Curve

Twisted Edwards curves were first described in [3] as a generalization of (ordinary) Edwards curves. Some of these curves are equipped with a very fast and complete addition law, whereby the rational point $\mathcal{O} = (0, 1)$ serves as neutral element. Completeness means that the addition law works for any two points P, Q that lie on the curve, including corner cases such as $P = \mathcal{O}$, $Q = \mathcal{O}$, and $P = -Q$ [3]. Formally, a twisted Edwards curve over a prime field $\mathbb{F}_p$ is defined by the equation

$$E_T : ax^2 + y^2 = 1 + dx^2y^2 \tag{2}$$

where $a, d \in \mathbb{F}_p$ and $ad(a - d) \neq 0$. As explained in [3], the completeness of the addition law depends on the two curve parameters. More precisely, when a is a square and d is a non-square in $\mathbb{F}_p$, then the addition law can be complete and have no exceptions. Completeness is a valuable property if one aims for a side-channel resistant implementation of scalar multiplication since the corner cases mentioned above do not need to be treated separately, which naturally leads to a regular execution profile and constant execution time.

The so-called extended coordinates presented by Hisil et al [15] allow for a particularly fast addition of points when the curve parameter $a = -1$. In this case, a mixed addition (i.e. an addition where one of the two points is given in projective coordinates and the other in affine coordinates) requires only seven multiplications in $\mathbb{F}_p$. Extended projective coordinates were originally proposed in [15] as a quadruple of the form (X, Y, T, Z), whereby the fourth coordinate $T = XY/Z$. In our implementation, we further split T up into the two factors E and H, i.e. we have $EH = T = XY/Z$, since this facilitates an optimization of the point doubling formulae. Consequently, we use a quintuple of the form (X, Y, E, H, Z) with $EH = T = XY/Z$ to represent a projective point and the triple (u, v, w) with $u = (x + y)/2$, $v = (y - x)/2$, and $w = dxy$ to represent an affine point. The computational cost for a (complete) mixed addition amounts to seven multiplications, three additions, and three subtractions in $\mathbb{F}_p$, while a projective doubling takes three multiplications, four squarings, four additions and two subtractions. We implemented the scalar multiplication according to the highly-regular fixed-base comb technique described in [21], which uses eight pre-computed points and processes four bits of the scalar at a time.

As demonstrated by Bernstein et al [3], every Montgomery curve over $\mathbb{F}_p$ is birationally equivalent over $\mathbb{F}_p$ to a twisted Edwards curve and vice versa. This equivalence is very useful in ephemeral ECDH key exchange since it allows one to perform the fixed-base scalar multiplication (for generating a key pair) on a twisted Edwards curve and the variable-base scalar multiplication (to compute the shared secret) on the birationally-equivalent Montgomery curve [21].

5 Results and Comparison

We complied and assembled the source code of our ECC software for MSP430 microcontrollers using version 6.10 of IAR Embedded Workbench, which comes

Table 2. Execution time and code size of arithmetic operations in 159, 191, 223, and 255-bit pseudo-Mersenne prime fields on a TI MSP430F1611 processor (all execution times include the full function-call overhead and the modular reduction; the values in parentheses indicate the time spent for the reduction operation alone).

Operation	Execution time (in clock cycles)				Code size
	159 bit	191 bit	223 bit	255 bit	(in bytes)
Addition	226	258	290	322	100
Subtraction	244	278	312	346	120
Multiplication	2448 (388)	3304 (452)	4288 (516)	5400 (580)	360 (168)
Squaring	1998 (388)	2578 (452)	3214 (516)	3914 (580)	406 (168)
32-bit Mul.	700 (232)	804 (258)	908 (284)	1012 (310)	282 (164)
Inversion	147440	202358	265318	336270	966

with a cycle-accurate instruction set simulator. Table 2 specifies the execution time and code size of the major operations of our parameterized $\mathbb{F}_p$-arithmetic library for 159, 191, 223 and 255-bit fields. The concrete primes with which we collected the simulation results are those from the four so-called MoTE curves specified in [10], namely $2^{159} - 91$, $2^{191} - 19$, $2^{223} - 235$, and $2^{255} - 19$. A full multiplication in a 159-bit pseudo-Mersenne prime field takes 2448 clock cycles altogether, to which the reduction contributes 388 cycles, i.e. the multiplication alone (without modular reduction) executes in 2160 cycles. Squaring (including reduction) is roughly 18.4% faster than multiplication. However, the difference between multiplication and squaring increases to some 27.5% in a 255-bit field (5400 versus 3914 cycles). Also provided in Table 2 is the execution time of the multiplication of a field element by a 32-bit integer; this operation can be used in e.g. the point doubling on a Montgomery curve for the multiplication by the constant $(A + 2)/4$. Inversion is the by far most expensive arithmetic operation in $\mathbb{F}_p$; a single inversion takes slightly more time than 60 multiplications. Note that all operations listed in Table 2, except inversion, have constant execution time. Since the number of clock cycles for an EAA-based inversion depends on the value of the operand to be inverted, we specify the average execution time in Table 2, which we found through inversion of 100 random field elements.

The last column in Table 2 summarizes the code size of different arithmetic functions of our library. The function for multiplying multiple-precision integers according to Algorithm 1 has a size of 192 bytes, while the modular reduction function occupies only 168 bytes in Flash memory, i.e. both together amounts to 360 bytes. Squaring is, in terms of code size, slightly larger than multiplication (by exactly 46 bytes) since it needs an extra loop. In general, our $\mathbb{F}_p$-arithmetic library is very compact because we avoided code-size increasing optimizations like loop unrolling. The size of the whole library amounts to about 2.3 kB; this includes besides the arithmetic operations also some auxiliary functions (e.g. to copy a field element or to check whether two field elements are equal).

We also evaluated the execution time of variable-base scalar multiplication (using the basic Montgomery ladder) on four different Montgomery curves and

Table 3. Execution time (in clock cycles on an MSP430F1611) of variable-base scalar multiplication on a Montgomery curve and fixed-base scalar multiplication on a twisted Edwards curve over 159, 191, 223, and 255-bit fields.

Curve type	159 bit	191 bit	223 bit	255 bit
Montgomery (variable base)	$3.86 \cdot 10^6$	$6.00 \cdot 10^6$	$8.79 \cdot 10^6$	$12.34 \cdot 10^6$
Twisted Edwards (fixed base)	$1.92 \cdot 10^6$	$3.01 \cdot 10^6$	$4.45 \cdot 10^6$	$6.29 \cdot 10^6$

fixed-base scalar multiplication (using a comb method with eight pre-computed points) on the four bitrationally equivalent twisted Edwards curve. The curves we used to obtain the simulation results are specified in [10]. As summarized in Table 3, the execution times for variable-base scalar multiplication range from $3.86 \cdot 10^6$ cycles (Montgomery curve over 159-bit field) up to $12.34 \cdot 10^6$ cycles (255-bit field). On the other hand, the fixed-base scalar multiplications (on the twisted Edwards curves) take only about one half of the execution time of the variable-base scalar multiplications at the same security level. In terms of code size, the C implementation of the point arithmetic on the Montgomery curve is about 1.7 kB; this includes besides point addition/doubling and scalar multiplication also a few auxiliary functions (e.g. for projective-to-affine conversion of a point). The code size of the point arithmetic on the twisted Edwards curve amounts to roughly 2.1 kB, again including some auxiliary functions. We have eight pre-computed points in extended affine coordinates per curve, which, in total (i.e. for four curves), occupy 2.5 kB in Flash memory.

In recent years, numerous papers on efficient ECC software for MSP430(X) processors have been published, e.g. [6, 9, 14, 20, 22, 27, 32, 37]. However, in the majority of these works, ordinary curves in Weierstraß form were used and the implementations lack protection against timing attacks, which makes it hard to compare the reported results with ours. Only Liu et al in [20] and Düll et al in [6] adopted Montgomery curves, but they entirely unrolled the field arithmetic (i.e. these implementations are not scalable). The former authors achieved an execution time of $3.25 \cdot 10^6$ and $5.12 \cdot 10^6$ clock cycles for scalar multiplication on a 159-bit and a 191-bit Montgomery curve, respectively, which outperforms our scalable ECC software by less than 20%. Düll et al reported $7.93 \cdot 10^6$ cycles for a scalar multiplication on Curve25519 and a code size of 13.1 kB, but these results were obtained using an MSP430FR5969 as evaluation platform. To aid comparison, we simulated our software with the parameters of Curve25519 and obtained an execution time of $10.85 \cdot 10^6$ clock cycles on the same device. Consequently, our scalable implementation is approximately 1.37 times slower than that of Düll et al, but more than three times smaller.

6 Conclusions

We presented the concept of energy scalability as an approach to minimize the total energy consumption of ECC operations in a WSN or, more generally, the IoT. Taking an ECC-based security architecture for a WSN as case study, we argued that node authentication has higher security requirements than e.g. the establishment of a shared secret key between nodes to encrypt short-lived data like sensor readings. By using elliptic-curve groups of smaller order for the less security-critical task(s), it is possible to save precious energy; for example, one could adopt a 191-bit curve for key establishment and a 223-bit curve for node authentication. We introduced a scalable yet efficient software implementation of ECC for 16-bit MSP430 processors that supports Montgomery and twisted Edwards curves. The core component of our ECC software is a parameterized library for arithmetic in pseudo-Mersenne prime fields, which is able to process operands of various lengths and has a binary code size of only 2.3 kB since we refrained from loop unrolling and other size-increasing optimizations. Nonetheless our software is only a factor of 1.37 slower than the high-speed Curve25519 implementation of Düll et al, but three times smaller. In summary, our results show that reaching good performance does not necessarily have to come at the expense of large code size and poor scalability.

Acknowledgments. Lin Li and Qiuliang Xu were supported by the National Natural Science Foundation of China under grant No. 61572294. This work was supported by the NSERC CREATE Training Program in Building a Workforce for the Cryptographic Infrastructure of the 21st Century (CryptoWorks21), and Public Works and Government Services Canada.

The full source code of the scalable ECC implementation described in this paper is available online at `http://www.cryptolux.org/index.php/SECC430`.

References

1. O. Alfandi, A. Bochem, A. Kellner, C. Göge, and D. Hogrefe. Secure and authenticated data communication in wireless sensor networks. *Sensors*, 15(8):19560–19582, Aug. 2015.
2. D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, eds., *Public Key Cryptography — PKC 2006*, vol. 3958 of *Lecture Notes in Computer Science*, pp. 207–228. Springer Verlag, 2006.
3. D. J. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters. Twisted Edwards curves. In S. Vaudenay, ed., *Progress in Cryptology — AFRICACRYPT 2008*, vol. 5023 of *Lecture Notes in Computer Science*, pp. 389–405. Springer Verlag, 2008.
4. C. Bormann, M. Ersue, and A. Keranen. Terminology for Constrained-Node Networks. Internet Engineering Task Force, Light-Weight Implementation Guidance Working Group, RFC 7228, May 2014.
5. D. Dang, M. Plant, and M. Poole. Wireless connectivity for the Internet of Things (IoT) with MSP430 microcontrollers (MCUs). Texas Instruments white paper, available for download at `http://www.ti.com/lit/wp/slay028/slay028.pdf`, Mar. 2014.
6. M. Düll, B. Haase, G. Hinterwälder, M. Hutter, C. Paar, A. H. Sánchez, and P. Schwabe. High-speed Curve25519 on 8-bit, 16-bit and 32-bit microcontrollers. *Designs, Codes and Cryptography*, 77(2–3):493–514, Dec. 2015.

7. D. Evans. The Internet of things: How the next evolution of the Internet is changing everything. Cisco IBSG white paper, available for download at `http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf`, Apr. 2011.
8. X. Fan and G. Gong. Accelerating signature-based broadcast authentication for wireless sensor networks. *Ad Hoc Networks*, 10(4):723–736, June 2012.
9. C. P. Gouvêa and J. López. Software implementation of pairing-based cryptography on sensor networks using the MSP430 microcontroller. In B. Roy and N. Sendrier, eds., *Progress in Cryptology — INDOCRYPT 2009*, vol. 5922 of *Lecture Notes in Computer Science*, pp. 248–262. Springer Verlag, 2009.
10. J. Großschädl. A family of implementation-friendly MoTE elliptic curves. Technical Report TR-LACS-2013-01, Laboratory of Algorithmics, Cryptology and Security (LACS), University of Luxembourg, Luxembourg, 2013.
11. V. Gupta, M. Wurm, Y. Zhu, M. Millard, S. Fung, N. Gura, H. Eberle, and S. Chang Shantz. Sizzle: A standards-based end-to-end security architecture for the embedded Internet. *Pervasive and Mobile Computing*, 1(4):425–445, Dec. 2005.
12. D. R. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Verlag, 2004.
13. T. Heer, O. Garcia-Morchon, R. Hummen, S. L. Keoh, S. S. Kumar, and K. Wehrle. Security challenges in the IP-based Internet of things. *Wireless Personal Communications*, 61(3):527–542, Dec. 2011.
14. G. Hinterwälder, A. Moradi, M. Hutter, P. Schwabe, and C. Paar. Full-size high-security ECC implementation on MSP430 microcontrollers. In D. F. Aranha and A. J. Menezes, eds., *Progress in Cryptology — LATINCRYPT 2014*, vol. 8895 of *Lecture Notes in Computer Science*, pp. 22–38. Springer Verlag, 2015.
15. H. Hişil, K. K.-H. Wong, G. Carter, and E. Dawson. Twisted Edwards curves revisited. In J. Pieprzyk, ed., *Advances in Cryptology — ASIACRYPT 2008*, vol. 5350 of *Lecture Notes in Computer Science*, pp. 326–343. Springer Verlag, 2008.
16. S. Kar. Cisco says Internet of things will have ten times more impact on society than Internet. Cloud Times, avialable online at `http://cloudtimes.org/2014/03/07/cisco-says-internet-of-things-will-have-ten-times-more-impact-on-society-than-internet`, Mar. 2014.
17. A. A. Karatsuba and Y. P. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics - Doklady*, 7(7):595–596, Jan. 1963.
18. P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. I. Koblitz, ed., *Advances in Cryptology — CRYPTO '96*, vol. 1109 of *Lecture Notes in Computer Science*, pp. 104–113. Springer Verlag, 1996.
19. Z. Liu, H. Seo, J. Großschädl, and H. Kim. Reverse product-scanning multiplication and squaring on 8-bit AVR processors. In L. C.-K. Hui, S. Qing, E. Shi, and S.-M. Yiu, eds., *Information and Communications Security — ICICS 2014*, vol. 8958 of *Lecture Notes in Computer Science*, pp. 158–175. Springer Verlag, 2015.
20. Z. Liu, H. Seo, Z. Hu, X. Huang, and J. Großschädl. Efficient implementation of ECDH key exchange for MSP430-based wireless sensor networks. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2015)*, pp. 145–153. ACM Press, 2015.
21. Z. Liu, E. Wenger, and J. Großschädl. MoTE-ECC: Energy-scalable elliptic curve cryptography for wireless sensor networks. In I. Boureanu, P. Owezarski, and S. Vaudenay, eds., *Applied Cryptography and Network Security — ACNS 2014*, vol. 8479 of *Lecture Notes in Computer Science*, pp. 361–379. Springer Verlag, 2014.

22. L. Marin, A. J. Jara, and A. F. Gómez-Skarmeta. Shifting primes: Extension of pseudo-Mersenne primes to optimize ECC for MSP430-based future Internet of things devices. In A. M. Tjoa, G. Quirchmayr, I. You, and L. Xu, eds., *Availability, Reliability and Security for Business, Enterprise and Health Information Systems*, vol. 6908 of *Lecture Notes in Computer Science*, pp. 205–219. Springer Verlag, 2011.
23. Memsic, Inc. TelosB Mote Platform. Data sheet, available for download at `http://www.memsic.com/userfiles/files/Datasheets/WSN/6020-0094-02_B_TELOSB.pdf`, Mar. 2007.
24. P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, Jan. 1987.
25. R. G. Moskowitz and R. Hummen. HIP Diet EXchange (DEX). Internet Engineering Task Force, Network Working Group, Internet draft, July 2015.
26. D. Naccache, N. P. Smart, and J. Stern. Projective coordinates leak. In C. Cachin and J. Camenisch, eds., *Advances in Cryptology — EUROCRYPT 2004*, vol. 3027 of *Lecture Notes in Computer Science*, pp. 256–267. Springer Verlag, 2004.
27. C. Pendl, M. Pelnar, and M. Hutter. Elliptic curve cryptography on the WISP UHF RFID tag. In A. Juels and C. Paar, eds., *RFID Security and Privacy — RFIDSec 2011*, vol. 7055 of *Lecture Notes in Computer Science*, pp. 32–47. Springer Verlag, 2012.
28. C. S. Raghavendra, K. M. Sivalingam, and T. F. Znati. *Wireless Sensor Networks*. Kluwer Academic Publishers, 2004.
29. E. K. Rescorla and N. G. Modadugu. Datagram Transport Layer Security Version 1.2. Internet Engineering Task Force, Network Working Group, RFC 6347, 2012.
30. R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, Feb. 1978.
31. A. Sehgal, V. Perelman, S. Kuryla, and J. Schönwälder. Management of resource constrained devices in the Internet of things. *IEEE Communications Magazine*, 50(12):144–149, Dec. 2012.
32. H. Seo, K.-A. Shim, and H. Kim. Performance enhancement of TinyECC based on multiplication optimizations. *Security and Communication Networks*, 6(2):151–160, Feb. 2013.
33. A. Sinha. *Energy Efficient Operating Systems and Software*. Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2001.
34. J. R. Smith, ed.. *Wirelessly Powered Sensor Networks and Computational RFID*. Springer Verlag, 2013.
35. F. Stajano, D. Cvrcek, and M. Lewis. Steel, cast iron and concrete: Security engineering for real world wireless sensor networks. In S. M. Bellovin, R. Gennaro, A. D. Keromytis, and M. Yung, eds., *Applied Cryptography and Network Security — ACNS 2008*, vol. 5037 of *Lecture Notes in Computer Science*, pp. 460–478. Springer Verlag, 2008.
36. Texas Instruments, Inc. MSP430x1xx Family User's Guide. Manual, available for download at `http://www.ti.com/lit/ug/slau049f/slau049f.pdf`, Feb. 2006.
37. E. Wenger and M. Werner. Evaluating 16-bit processors for elliptic curve cryptography. In E. Prouff, ed., *Smart Card Research and Advanced Applications — CARDIS 2011*, vol. 7079 of *Lecture Notes in Computer Science*, pp. 166–181. Springer Verlag, 2011.
38. Y. Zhou, Y. Zhang, and Y. Fang. Access control in wireless sensor networks. *Ad Hoc Networks*, 5(1):3–13, Jan. 2007.