Energy-Scalable Montgomery-Curve ECDH Key Exchange for ARM Cortex-M3 Microcontrollers

Christian Franck, Johann Großschädl, Yann Le Corre, and Cyrille Lenou Tago
University of Luxembourg, Computer Science and Communications Research Unit
6, Avenue de la Fonte, L–4364 Esch-sur-Alzette, Luxembourg
{christian.franck,johann.groszschaedl,yann.lecorre}@uni.lu
cyrille.lenou.001@student.uni.lu

***Abstract*—The number of smart devices connected to the Internet is growing at an enormous pace and will reach 30 billion within the next five years. A large fraction of these devices have limited processing capabilities and energy supply, which makes the execution of computation-intensive cryptographic algorithms very costly. This problem is exacerbated by the fact that basic optimization techniques like loop unrolling can not (always) be applied since cryptographic software for the IoT often needs to meet strict constraints on code size to not exceed the program storage capacity of the target device. In this paper we introduce SECCCM3, a "lightweight" software library for scalable elliptic curve cryptography on ARM Cortex-M3 microcontrollers. The current version of SECCCM3 is able to carry out variable-base scalar multiplication on Montgomery-form curves over pseudo-Mersenne prime fields, such as Curve25519, and can be used to implement static ECDH key exchange. SECCCM3 is scalable in the sense that it supports curves of different order (as long as certain conditions are met), thereby enabling trade-offs between security and execution time (resp. energy dissipation). We made an effort to protect the field arithmetic against Timing Attacks (TAs) and Simple Power Analysis (SPA), taking into account the so-called early-termination effect of the Cortex-M3 integer multiplier, which makes the latency of "long" multiply instructions operand-dependent. Our experiments show that the integration of countermeasures against information leakage caused by this effect increases the execution time by 34%, while the code size grows by 13%. A TA and SPA-resistant scalar multiplication on Curve25519 has an execution time of 4.565 million clock cycles and consumes approximately 5.1 mJ of energy when executed on a STM32L152RE Cortex-M3 microcontroller. SECCCM3 has a binary code size of 4.0 kB, which includes domain parameters for curves over 159, 191, 223, and 255-bit prime fields.**

***Index Terms*—Elliptic curve cryptography, Pseudo-Mersenne prime, Modular arithmetic, ARM Cortex-M3 early-termination effect, Constant-time multi-precision multiplication.**

I. Introduction

The Internet of Things (IoT) is the natural next step in the evolution of the Internet towards a network to which billions of "smart" devices (i.e. "things" with computing capabilities) get connected, enabling them to send and receive data [1]. In this way, the devices can interact with each other and access centralized resources and services in the cloud. The Ericsson Mobility Report from November 2017 [9] forecasts the total number of devices connected to the Internet to increase from slightly less than 20 billion in 2018 to more than 30 billion within the next five years (i.e. until 2023). It is remarkable that only about one third of these 30 billion devices can be counted as classical computers (i.e. PCs, notebooks, mobile phones, etc), while the remaining two third represent various kinds of smart devices (i.e. computers that do not look like computers), such as vehicles, machines, meters, sensors, point-of-sale terminals, consumer electronics, and wearables. The majority of these IoT devices are significantly constrained in terms of computing power, RAM, and program storage. This is especially the case for battery-driven devices like wireless sensor nodes, which are typically equipped with inexpensive 8, 16, or 32-bit microcontrollers clocked with a few MHz. In addition, the RAM and program-storage (i.e. flash) capacities of sensor nodes and many other IoT devices are quantified in kilobytes rather than megabytes or gigabytes.

TABLE I
Elliptic Curves Used in IoT Communication Protocols

NFC Forum [18] NFC Signature RTD	IEEE VT Society [13] Vehicular Adhoc Netw.	Bluetooth SIG [4] Bluetooth Low Energy
secp192r1, secp224r1, sect233r1, sect233k1, secp256r1	secp224r1, secp256r1	secp192r1, secp256r1

The resource constraints of common IoT devices make the implementation of cryptographic algorithms challenging; this is particularly the case for public-key schemes. Elliptic curve cryptosystems, such as ECDSA and ECDH [5], are attractive alternatives to RSA and Diffie-Hellman [7] since they provide a higher level of security per key-bit, which enables them to achieve faster execution times and lower RAM consumption [12]. Recently, a number of organizations concerned with the standardization of (wireless) communication protocols for the IoT (including e.g. the Bluetooth Special Interest Group, the IEEE Vehicular Technology Society, and the NFC Forum, to name a few) have adopted elliptic curve cryptography for the creation of shared secret keys and the generation/verification of digital signatures. Table I gives references to the protocol specifications and shows an overview of the deployed curves using the nicknames of the Standards for Efficient Cryptography Group (SECG) defined in [20]. Each protocol supports several (at least two) elliptic curves providing different levels of security. The specification of the NFC Forum's Signature Record Type Definition (RTD) [18] comes with a total of five curves that represent security levels from 96 to 128 bits.

Designing communication protocols for the IoT in an agile fashion so that they can support multiple key lengths for the used cryptographic algorithms makes a lot of sense because different applications typically have different security requirements and impose different constraints on the execution time and resource consumption (RAM, flash, etc). An application for home automation, e.g. a sensor network that monitors the temperature in an apartment to control a HVAC system, has completely different security requirements than applications in the medical or automotive domain. It is, of course, possible to satisfy different security requirements with a single elliptic curve, namely by simply choosing the "strongest" curve the protocol supports (which is secp256r1 for the three protocols in Table I), but doing so entails unnecessarily long execution times and wastes scarce memory resources if an application has only low or medium security requirements. To minimize such wastage, IoT application designers normally choose the curve that guarantees the required level of security with the least impact on the latency of the protocol and its resource consumption. Most state-of-the-art IoT devices that are able to perform elliptic curve cryptosystems support only a single curve, which is typically one of the curves in [20].

Supporting a single elliptic curve is sufficient for devices used by current-generation IoT applications, but not for the future IoT. In the present-day IoT, the main communication pattern is often device-to-cloud; a typical example is a device that sends data to a cloud service for analysis and decision making. On the other hand, device-to-device communication does normally only take place between devices belonging to the same application, e.g. the nodes of a wireless sensor network. Indeed, in the current IoT, there is not much interaction between devices of different applications, in particular if the devices come from different, competing vendors and run on different, often incompatible platforms. However, in order to fully realize the potential of the future IoT, it is inevitable that diverse and heterogenous IoT devices, which may form part of different applications from different vendors, become able to communicate and interact with each other, either directly or via some kind of gateway[1]. This vision of a future IoT with highly interoperable devices calls for a flexible and scalable implementation of elliptic curve cryptography that is able to handle many different curves with minimal impact on code size. Such an implementation would allow the devices of an application with low security requirements, which can use e.g. secp192r1, to perform ECDH exchange with devices of a high-security application using e.g. secp256r1.

In this paper, we present a lightweight and highly scalable software library for variable-base scalar multiplication, called SECCCM3 (an abbreviation of Scalable Elliptic Curve Cryptography for Cortex-M3), which provides the full arithmetic functionality needed for ECDH key exchange. As its name suggests, SECCCM3 is optimized for the Cortex-M3 series [21] of ARM microcontrollers and comes with hand-written Assembly code for the low-level prime-field arithmetic so as to improve the execution time and reduce the code size. The current version of SECCCM3 supports elliptic curves given in Montgomery form [17] because they offer computational advantages over the Weierstrass curves specified in [20]. We implemented the multiplication and squaring operation in the underlying prime field $\mathbb{F}_p$ to have constant execution time so that SECCCM3 can resist timing attacks, which is far from trivial on Cortex-M3 processors due to the early-termination effect of the integer multiplier [11]. SECCCM3 enables the realization of energy-scalable ECDH key exchange since one can (dynamically) switch between curves of different orders that provide different security levels, have different execution times, and consume different amounts of energy. We analyze the performance and energy figures of scalar multiplication for four different security levels ranging from 80 to 128 bits and compare our results with that of some other libraries.

[1]Such IoT gateways typically support multiple (wireless) communication protocols, e.g. Bluetooth, WiFi, and ZigBee. Note that also a smart phone or tablet computer can serve as gateway for devices.

II. Background

The ECDH key exchange protocol [5] enables two parties communicating over a public network to establish a common shared secret S, which is, in our case, simply the x-coordinate of a point on an elliptic curve. Let us assume that Alice and Bob have the private key a and b, respectively, and that these keys are integers in the range of $[1, n-1]$, where n is prime and denotes the order of an elliptic-curve (sub)group generated by a certain point G. For sufficiently large order n, the computation of the scalar x for a point $P = xG$ is assumed to be a hard mathematical problem [5]. When Alice publishes aG and Bob publishes bG, then each of them can compute the secret point $S = a(bG) = b(aG)$, but an adversary who has only access to aG and bG can not obtain S since he does not know the secrets a and b (which he also can not compute from aG and bG when G has large order).

A Montgomery curve [17] over a non-binary field $\mathbb{F}_q$ is an elliptic curve governed by an equation of the form

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

where $A, B \in \mathbb{F}_q$ and $(A^2 - 4)B \neq 0$. Although these curves were originally proposed for integer factorization, it became more and more apparent in the past ten years that they also have some advantages for ECDH key exchange [3]. The main attraction of Montgomery-form curves is a special algorithm for a variable-base scalar multiplication $Q = kP$, the Montgomery ladder [5], which is not only fast but also modest in terms of RAM consumption and has as further benefit some intrinsic resistance against timing attacks as well as Simple Power Analysis (SPA). Montgomery curves feature a unique addition law that allows one to efficiently compute the sum $P_1 + P_2$ of two points P_1, P_2 whose difference $P_1 - P_2$ is known at the cost of only three multiplications (3M) and two squarings (2S) in $\mathbb{F}_q$ when using projective coordinates. This so-called "differential" addition involves only the projective X and Z coordinates, i.e. the Y coordinate is not used at all [17]. The doubling of a point in projective X, Z coordinates is even cheaper and costs only $2M$ and $2S$. In summary, the

Montgomery ladder has to carry out $5n$ multiplications and $4n$ squarings in $\mathbb{F}_q$ to compute $Q = kP$ when the scalar k is n bits long, i.e. $5M$ and $4S$ per bit.

As demonstrated in [3], Montgomery curves are attractive for ECDH key exchange since only the x-coordinate of the point representing a public key needs to be transmitted when the curve has been chosen properly. More concretely, if the curve is twist-secure, such as Curve25519 [3], then one does not need to care whether a received x-coordinate belongs to a point on the curve or on the twist, i.e. it is not necessary to validate public keys. In this way, it is possible to perform the computation and communication part of ECDH with just the x-coordinates of the points, which means point compression (and costly decompression) is not necessary.

III. Implementation

SECCCM3 supports Montgomery curves over prime fields that are defined by pseudo-Mersenne primes [5] of the form $p = 2^k - c$ where k is a multiple of 32 minus 1 and c has a length of up to 14 bits. The four specific curves we used to generate the benchmarking results presented in this paper are specified in [10]; they are based on pseudo-Mersenne prime fields of a size of 159, 191, 223, and 255 bits. The curves in [10] were specifically designed to provide consistency across security levels (i.e. the curves and fields share various basic properties), which facilitates a parameterized implementation of the low-level field arithmetic and scalar multiplication. All implementations of $\mathbb{F}_p$-arithmetic functions described in this section are parameterized in the sense that they get besides the "actual" operands (which are typically pointers to arrays of 32-bit words in RAM) a further parameter that specifies the length of the operands, i.e. the number of 32-bit words they consist of. Each function contains one or more loops to perform an arithmetic operation, whereby the number of loop iterations depends on the value of the length parameter. This approach allows one and the same software implementation of an arithmetic function to support pseudo-Mersenne prime fields of (almost) arbitrary order, provided the prime p meets the restrictions on k and c mentioned above.

A. Prime-Field Arithmetic

SECCCM3 comes with an efficient and scalable arithmetic library for pseudo-Mersenne prime fields that was written in Assembly language for the Cortex-M3 platform. We aimed to achieve a balance between flexibility, performance, and code size instead of optimizing purely for high speed, which was the goal of most existing implementations. Our intention was to have a scalable implementation that allows us to study the (relative) impact of the security level on execution time and energy consumption rather than setting new speed records.

Representation of field elements: The elements of a prime field $\mathbb{F}_p$ are integers between 0 and $p - 1$. We represent the field elements through arrays of 32-bit words (i.e. unsigned 32-bit integers), whereby in our case the number of words amounts to $l = \lceil k/32 \rceil$ because our primes are of the form $p = 2^k - c$. For example, when p is a 255-bit prime such as $2^{255} - 19$, then a field element consists of eight words. Note that the functions of our arithmetic library do not insist the operands to be less than p but accept incompletely-reduced operands as input, as long as they fit into l words. Also the results produced by the functions of our arithmetic library can be slightly larger than p, but they always fit into l words.

Addition: An addition of two elements $a, b \in \mathbb{F}_p$ consists of computing the sum $r = a + b$ and then reducing r modulo p. Since both a and b can be incompletely reduced, the sum r may be up to $32l + 1 = k + 2$ bits long (recall that k is a multiple of 32 minus 1, i.e. $k = 32l - 1$). The reduction operation can take advantage of the special form of p, i.e. we have $2^k \equiv c \bmod p$. In order to reduce the $(k + 2)$-bit sum r modulo p, we split r up into a lower part r_L that consists of the k least significant bits of r and a higher part r_H of up to two bits in length. Consequently, the sum r can be written as $r = r_H 2^k + r_L$ and we can simply substitute 2^k by c to obtain a result of a length of at most $k + 1$ bits that fits into l words. Our implementation follows the approach described in [8] and starts the addition with the most significant words of a and b, which means a_{l-1} and b_{l-1} are added first. The two most significant bits of this sum (which can be up to 33 bits long) represent r_H and are multiplied by c to obtain an up to 16 bits long product. Then, the remaining 32-bit words of the operands a and b are summed up, beginning with the two least significant words a_0 and b_0, to which also $r_H c$ is added. The sum of a_0, b_0, and $r_H c$ can have a length of up to 34 bits and, therefore, the carries that propagate from less to more significant words are in the range of $[0, 2]$.

Subtraction: The standard way to perform a subtraction in $\mathbb{F}_p$ consists of carrying out an ordinary subtraction and then conditionally adding p if the difference was negative. While this approach is easy to implement, it consists of two loops (i.e. high loop overhead) and has operand-dependent execution time. To resolve these issues, we add a multiple of p to the difference, i.e. we realize the subtraction as an operation of the form $r = 4p + a - b = 2^{k+2} + a - b - 4c \bmod p$. The addition of $4p$ does not introduce a lot of overhead because adding 2^{k+2} and subtracting $4c$ only affects the computation of the most and least significant word, respectively. Like the $\mathbb{F}_p$-addition described above, also the $\mathbb{F}_p$-subtraction requires a modular reduction, which can be performed in basically the same way. The processing starts at the most significant word of a and b, and then the main loop is executed, whereby the carries propagating from less to more significant words are in the range of $[-2, 1]$. To ensure that the final result is non-negative, we actually compute the most significant word as $r_{l-1} = 2^{33} - 2 + a_{l-1} - b_{l-1}$, then extract r_H (which can be up to three bits long) from it, and finally add 2 to guarantee $r_{l-1} \geq 2$ so that a negative carry can be absorbed.

Multiplication: The most basic technique to multiply two elements $a, b \in \mathbb{F}_p$ consists of a normal integer multiplication $t = ab$, followed by a reduction of the product t modulo the prime p. From an algorithmic point of view, there exist two major approaches to implement a long-integer multiplication in software, namely the Operand-Scanning (OS) method and

the Product-Scanning (PS) method [12]. These two methods differ in their loop structure and the operation they execute in the inner loops. Cortex-M models based on the ARMv7E-M architecture, such as Cortex-M4 and Cortex-M7, support the `umaal` instruction, which executes exactly the operation carried out in the inner loop of the OS method. While `umaal` is not available on Cortex-M3 processors, the `umlal` instruction is supported; it multiplies the content of two 32-bit registers and adds the 64-bit product to a 64-bit accumulator held in two 32-bit registers. This is, in principle, the operation in the inner loops of the PS method, but the 64-bit accumulator is a serious limitation that makes `umlal` little useful for long-integer multiplication unless one reduces the number of bits per word from 32 to e.g. 28. A reduced-radix representation with 28-bit words allows several word-products to be added into a 64-bit accumulator without overflow, but increases the number of iterations of the inner loop from l^2 to $(l+1)^2$. In accordance with the goals of SECCCM3, i.e. scalability and a "good" trade-off between execution time and code size, we decided to implement the multiplication using the PS method with a canonical (i.e. 32 bits per word) representation of the field elements, similar as described in [15]. In each iteration of the inner loop, two `ldr`, an `umull`, an `adds`, an `adcs`, as well as an `adc` instruction are executed.

The $2l$-word product $t = ab$ has to be reduced modulo p to get a result r that consists of l words. Our implementation of the reduction operation takes advantage of the special form of the prime; in particular, we exploit that $2^k \equiv c \bmod p$ and $2^{k+1} \equiv 2^{32l} \equiv 2c \bmod p$. The first step of the reduction is to multiply the l upper words of t by $2c$ and add the product to the l lower words, which yields an intermediate result t' of a length of $l+1$ words since $c \leq 2^{14} - 1$. Then, t' is split into a lower part t'_L comprising the k least significant bits of the intermediate result and a higher part t'_H that is up to 16 bits long. The final step is the multiplication of t'_H by c and the addition of $t'_H c$ to the lower part t'_L, which is similar to the last step of the $\mathbb{F}_p$-addition described above.

Squaring: Our implementation of the squaring takes into account that the square $t = a^2$ of an l-word integer a can be computed using only $(l^2 + l)/2$ `umull` instructions, which is significantly less than the l^2 `umull` instructions required to multiply two distinct l-word integers. We refer to [15] for a more detailed explanation of the squaring algorithm.

B. Constant-Time Multiplication

A non-trivial problem when implementing multi-precision arithmetic for the Cortex-M3 platform is that the instructions for (32×32)-bit multiplication yielding a 64-bit result have operand-dependent latency. For example, the execution of an `umull` instruction takes between three and five clock cycles depending on the actual bitlength of the operands [21]. This micro-architectural effect is called early termination and can make cryptographic software susceptible to Timing Analysis (TA) and SPA attacks [11]. To prevent such attacks, one has to perform all (32×32)-bit multiplications in constant time with minimum overhead. One approach to achieve this is to prepare the operands such that the multiplication instruction is forced to take a fixed number of cycles. Unfortunately, the early termination mechanism of the M3 is very complex (see [6]), which means simple techniques like the one proposed in [2] for an ARM7TDMI-S core will not work properly.

LISTING I
CONSTANT-TIME (32×32)-BIT MULTIPLICATION (12 CYCLES)

```
UMULL_CT(RLo,RHi,A,B):
    uxth ALo, A                 // prepare 16 bit
    uxth BLo, B
    mov  AHi, A, LSR 16
    mov  BHi, B, LSR 16
    mul  RLo, ALo, BLo          // lo x lo
    mul  RHi, AHi, BHi          // hi x hi
    mul  Tmp, ALo, BHi          // lo x hi
    adds RLo, RLo, Tmp, LSL 16
    adc  RHi, RHi, Tmp, LSR 16
    mul  Tmp, AHi, BLo          // hi x lo
    adds RLo, RLo, Tmp, LSL 16
    adc  RHi, RHi, Tmp, LSR 16
```

In order to make the prime-field arithmetic less vulnerable to TA and SPA attacks, we decided to use only constant-time instructions, which means we replaced the `umull` instruction for (32×32)-bit multiplication by the instruction sequence specified in Listing I. This sequence contains four (16×16)-bit multiplications executed by the `mul` instruction that takes exactly one cycle. Similar approaches for the Cortex-M0 are described in [8], [16], but our code is faster on a Cortex-M3 since the M3 is less restrictive regarding register usage. The computation takes 12 clock cycles and temporarily uses five additional registers for `ALo`, `AHi`, `BLo`, `BHi`, and `Tmp`.

LISTING II
CONSTANT-TIME (32×16)-BIT MULTIPLICATION (8 CYCLES)

```
UMULL2_CT(RLo,RHi,A,B):
    uxth ALo, A                 // prepare 16 bit
    uxth BLo, B
    mov  AHi, A, LSR 16
    mul  RLo, ALo, BLo          // lo x lo
    mul  Tmp, AHi, BLo          // hi x lo
    adds RLo, RLo, Tmp, LSL 16
    mov  RHi, 0
    adc  RHi, RHi, Tmp, LSR 16
```

LISTING III
CONSTANT-TIME 32-BIT SQUARING (9 CYCLES)

```
USQRL_CT(RLo,RHi,A):
   uxth ALo, A                  // prepare 16 bit
   mov  AHi, A, LSR 16
   mul  RLo, ALo, ALo           // lo x lo
   mul  RHi, AHi, AHi           // hi x hi
   mul  Tmp, ALo, BHi           // lo x hi
   adds RLo, RLo, Tmp, LSL 16
   adc  RHi, RHi, Tmp, LSR 16
   adds RLo, RLo, Tmp, LSL 16
   adc  RHi, RHi, Tmp, LSR 16
```

The number of instructions and temporarily-used registers can be reduced in some situations. For example, a (32×16)-bit multiplication, which is useful for the reduction modulo p, can be executed in eight cycles with two extra registers as shown in Listing II. The computation of a 32-bit square (see Listing III) requires nine cycles and two extra registers.

IV. Experimental Results

In this section, we present implementation results for three different versions of SECCCM3: (i) a highly portable version written in C, (ii) an optimized version for Cortex-M3 where the field arithmetic is written in assembler, and (iii) a secure version for Cortex-M3 where the field arithmetic is written in assembler using constant-time multiplication. SECCCM3 has not been specifically optimized for high performance but we rather tried to find a good trade-off between speed and code size. In particular, we did not unroll any of the loops of the field arithmetic operations to ensure SECCCM3 satisfies the code-size constraints of IoT devices. The portable C version and the simple assembler version for Cortex-M3 succumb to TA and SPA attacks because they execute non-constant-time multiplication instructions. The secure version for Cortex-M3 implements all (32×32)-bit multiplications as described in Section III-B and is, thus, protected against these attacks.

The portable C version of SECCCM3 has a (binary) code size of 5720 B; the two mixed C and assembler variants are smaller and amount to 3544 B for the basic version with the `umull` instruction and 4012 B for the secure (i.e. constant-time) version. Consequently, the integration of constant-time code as replacement for the variable-time `umull` instruction increases the code size of SECCCM3 by about 13.2%. This code size is still relatively small, especially when taking into account that parameters of four different Montgomery curves are included. For comparison, the X25519 implementation in [6] has a size of 4140 B, but supports just a single curve and is vulnerable to TA and SPA when executed on a Cortex-M3 processor. The prime-field multiplication of SECCCM3 has a size of 334 B when the `umull` instruction is used and 550 B when `umull` is replaced by constant-time instructions.

A. Execution Time

We evaluated our SECCCM3 library on a STM32 Nucleo board equipped with a STM32L152RE Cortex-M3 microcontroller, which we clocked with a frequency of 8.0 MHz. The functions we benchmarked were executed from flash with 0 wait states. Table II specifies the execution time of a single $\mathbb{F}_p$-multiplication for four fields. The basic assembler version using `umull` is roughly 8.5% faster than the C code, whereas the secure assembler variant is about 27-28% slower than the C code. When comparing the two assembler implementations directly we see that the proposed constant-time multiplication introduces a performance penalty of around 38-40%.

TABLE II
Computation time of a Prime-Field Multiplication

Field size	C Code	ASM	ASM (sec.)
159-bit field	717 cycles	657 cycles	911 cycles
191-bit field	957 cycles	879 cycles	1226 cycles
223-bit field	1235 cycles	1133 cycles	1585 cycles
255-bit field	1551 cycles	1419 cycles	1988 cycles

The execution time of a scalar multiplication, which is the main operation of ECDH, is shown in Table III. We observe that scalar multiplication on a 255-bit curve takes over three times longer than on a 159-bit curve. The "pure" C versions need 17-20% more execution time than the C code with the basic assembler implementation of the field arithmetic, but is around 8-14% faster than the C code with the constant-time arithmetic written in assembler. This mixed version using the secure arithmetic is between 30% and 34% slower than the mixed version with the non-constant-time assembler code.

TABLE III
Execution Time of a Scalar Multiplication

SECCCM3	C Code	C+ASM	C+ASM (sec.)
159-bit curve	1322678 cycles	1104006 cycles	1436706 cycles
191-bit curve	2005360 cycles	1690921 cycles	2232311 cycles
223-bit curve	2884341 cycles	2445445 cycles	3262689 cycles
255-bit curve	3992944 cycles	3396699 cycles	4565428 cycles
Micro-ECC		C+ASM	
160-bit curve		1868312 cycles	
192-bit curve		2369469 cycles	
224-bit curve		3569080 cycles	
256-bit curve		6817998 cycles	
De Groot [6]		ASM	
255-bit curve		2661659 cycles	
Nishinaga [19]			C+ASM (sec.)
255-bit curve			6245448 cycles

Besides SECCCM3, we also benchmarked Micro-ECC, an open-source library for elliptic curve cryptography written in C with "inlined" assembler code for ARM[2]. Micro-ECC can perform a variable-base scalar multiplication on five of the curves specified in [20]. We see in Table III that SECCCM3 compares well with Micro-ECC since, for similar curves, it is 10-20% faster and it furthermore is protected against attacks exploiting the early termination effect. De Groot presents in [6] a carefully tuned implementation of scalar multiplication on Curve25519 that uses Karatsuba's technique [14] to speed up the multiplication in the underlying prime field, which is an optimization not (yet) supported by SECCCM3. The full scalar multiplication takes only 2661659 clock cycles on the Cortex-M3, but is vulnerable to early-termination attacks. In [19], Nishinaga and Mambo ported μNaCl to the Cortex-M3 and managed to perform a full 255-bit scalar multiplication in 6245448 cycles, which is 36% slower than SECCCM3.

B. Energy Consumption

In our experiments, we supplied the STM32 Nucleo board with an external voltage of $V_{\mathrm{dd}} = 3.3$ V and an internal voltage of $V_{\mathrm{core}} = 1.8$ V. To determine the energy consumption we used a setup as shown in Figure 1. During the execution of a scalar multiplication on the M3, we measured a voltage of $V_{\mathrm{m}} = 17$ mV on a 6.2 Ω resistor, which means we have a current of $I_{\mathrm{dd}} = 2.74$ mA (this is very close to the typical current of 2.2 mA given in the data sheet). Thus, the M3 has a power consumption of $I_{dd} \cdot (V_{\mathrm{dd}} - V_{\mathrm{m}}) = 9.0$ mW.

[2]Micro-ECC is available on GitHub at https://github.com/kmackay/micro-ecc. We compiled it for the Cortex-M3 using the options Thumb2, native little endian, fast squaring, and optimization level 4.

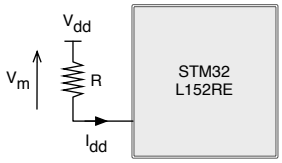

Fig. 1. Setup used for power measurement

When we combine these 9.0 mW with the cycle counts in Table III (whereby the clock frequency of 8 MHz needs to be taken into account), we obtain the energy required for the computation of a scalar multiplication, which is specified in Table IV. Unsurprisingly, the energy values are closely tied to the execution times. Taking the secure (i.e. constant-time) implementations as example, we observe the energy cost of a scalar multiplication to range from 1.62 mJ to 5.14 mJ. The 255-bit elliptic curve is in terms of energy over three times more expensive than the 159-bit curve. Note that the overall energy consumption of the ECDH protocol includes besides the computation energy for the scalar multiplication also the communication energy consumed for the transmission of the public key, which is outside the scope of this paper.

TABLE IV
ENERGY CONSUMPTION PER SCALAR MULTIPLICATION ON STM32

SECCCM3	C Code	C+ASM	C+ASM (sec.)
159-bit curve	1.49 mJ	1.24 mJ	1.62 mJ
191-bit curve	2.26 mJ	1.90 mJ	2.51 mJ
223-bit curve	3.25 mJ	2.75 mJ	3.67 mJ
255-bit curve	4.49 mJ	3.82 mJ	5.14 mJ
Micro-ECC		C+ASM	
160-bit curve		2.10 mJ	
192-bit curve		2.67 mJ	
224-bit curve		4.02 mJ	
256-bit curve		7.67 mJ	
De Groot [6]		ASM	
255-bit curve		2.99 mJ	
Nishinaga [19]			C+ASM (sec.)
255-bit curve			7.03 mJ

V. CONCLUDING REMARKS

We introduced SECCCM3, a lightweight cryptographic library to compute scalar multiplication on Montgomery-form elliptic curves of different order. All arithmetic operations in the underlying pseudo-Mersenne prime fields were written in ARM assembly language and optimized for the Cortex-M3 platform with the goal to achieve a sensible balance between scalability, performance, and code size. SECCCM3 includes domain parameters for elliptic curves over 159, 191, 223 and 255-bit prime fields, whereby the parameters can be adapted dynamically (i.e. at run-time without the need to re-compile the software) to guarantee high scalability and enable various trade-offs between security and energy consumption.

A non-trivial problem with Cortex-M3 microcontrollers is that the latency of long multiply instructions (e.g. `umull`) is not constant but depends on the value of the operands. This variable latency can make cryptographic software vulnerable to timing and SPA attacks, even if it was specifically written to be able to resist such attacks. We presented a constant-time multiplication routine for 32-bit operands that executes in 12 cycles on the Cortex-M3 and forms the basis for our secure implementation of the prime-field arithmetic.

As future work, we plan to speed up the $\mathbb{F}_p$-multiplication through the integration of Karatsuba's technique and analyze the communication energy cost of ECDH key exchange.

REFERENCES

[1] L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, Oct. 2010.
[2] F. Ben Hamouda. Exploration of effciency and side-channel security of different implementations of RSA. Technical report, available for download at http://www.normalesup.org/~fbenhamo/files/stage2011/report.pdf, 2011.
[3] D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In *Public Key Cryptography — PKC 2006*, vol. 3958 of *Lecture Notes in Computer Science*, pp. 207–228. Springer Verlag, 2006.
[4] Bluetooth Special Interest Group. Specification of the Bluetooth System (Version 4.2). Available for download at http://www.bluetooth.com/specifications/bluetooth-core-specification, 2014.
[5] H. Cohen and G. Frey. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Chapmann & Hall\CRC, 2006.
[6] W. de Groot. A performance study of X25519 on Cortex-M3 and M4. M.Sc. thesis, Eindhoven University of Technology, 2015.
[7] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.
[8] 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, 2015.
[9] Ericsson. Ericsson Mobility Report November 2017. Available for download at http://www.ericsson.com/assets/local/mobility-report/documents/2017/ericsson-mobility-report-november-2017.pdf, 2017.
[10] J. Großschädl. A family of implementation-friendly MoTE elliptic curves. Technical Report TR-LACS-2013-01, Laboratory of Algorithmics, Cryptology and Security, University of Luxembourg, 2013.
[11] J. Großschädl, E. Oswald, D. Page, and M. Tunstall. Side-channel analysis of cryptographic software via early-terminating multiplications. In *Information Security and Cryptology — ICISC 2009*, vol. 5984 of *Lecture Notes in Computer Science*, pp. 176–192. Springer Verlag, 2009.
[12] D. R. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Verlag, 2004.
[13] IEEE Vehicular Technology Society. IEEE Std 1609.2-2013: IEEE Standard for Wireless Access in Vehicular Environments—Security Services for Applications and Management Messages. Available for download at http://standards.ieee.org/findstds/standard/1609.2-2013.html, 2013.
[14] A. A. Karatsuba and Y. P. Ofman. Multiplication of multidigit numbers on automata. *Doklady Akademii Nauk SSSR*, 145(2):293–294, 1962.
[15] Z. Liu, J. Großschädl, L. Li, and Q. Xu. Energy-efficient elliptic curve cryptography for MSP430-based wireless sensor nodes. In *Information Security and Privacy — ACISP 2016*, vol. 9722 of *Lecture Notes in Computer Science*, pp. 94–112. Springer Verlag, 2016.
[16] Z. Liu, H. Seo, A. Castiglione, K.-K. R. Choo, and H. Kim. Memory-efficient implementation of elliptic curve cryptography for the internet-of-things. *IEEE Transactions on Dependable and Secure Computing* (to appear).
[17] P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.
[18] NFC Forum. NFC Signature RTD Technical Specification 2.0 and Certificate Policy. Available for download at http://nfc-forum.org/product/nfc-signature-rtd-certificate-policy, 2015.
[19] T. Nishinaga and M. Mambo. μNaCl on 32-bit ARM Cortex-M3. In *Proceedings of the 2015 Computer Security Symposium*, pp. 102–109, 2015.
[20] Standards for Efficient Cryptography Group (SECG). SEC 2: Recommended Elliptic Curve Domain Parameters (Version 2). Available for download at http://www.secg.org/sec2-v2.pdf, 2010.
[21] J. Yiu. *The Definitive Guide to the ARM Cortex-M3*. Newnes, 2009.