

Enabling Full-Size Public-Key Algorithms on 8-Bit Sensor Nodes

Leif Uhsadel, Axel Poschmann, and Christof Paar

Horst Görtz Institute for IT Security
Communication Security Group (COSY)
Ruhr-Universität Bochum, Germany
Universitätsstrasse 150
44780 Bochum, Germany
{uhsadel, poschmann, cpaar}@crypto.rub.de
www.crypto.rub.de

Abstract. In this article we present the fastest known implementation of a modular multiplication for a 160-bit standard compliant elliptic curve (secp160r1) for 8-bit micro controller which are typically used in WSNs. The major part (77%) of the processing time for an elliptic curve operation such as ECDSA or EC Diffie-Hellman is spent on modular multiplication. We present an optimized arithmetic algorithm which significantly speed up ECC schemes. The reduced processing time also yields a significantly lower energy consumption of ECC schemes. With our implementation results we can show that a 160-bit modular multiplication can be performed in 0.39 ms on an 8-bit AVR processor clocked at 7.37 MHz. This brings the vision of asymmetric cryptography in the field of WSNs with all its benefits for key-distribution and authentication a step closer to reality.

Keywords: wireless sensor network, elliptic curve cryptography, 8-bit micro controller, Micaz, secp160r1.

1 Introduction

The terms *ubiquitous* and *pervasive computing* designate the penetration of our everyday life with intelligent devices. *Wireless sensor networks* (WSN) will play a fundamental role to enable this vision. WSNs consist of many tiny and smart devices, referred to as nodes, which typically combine an 8-bit processor with memory, sensors, radio unit and power supply. The foreseen applications for WSNs range from medical scenarios to agricultural, military and environmental monitoring. Since many data may be very critical (e.g., for the health of human beings in medical scenarios or safety critical monitoring) security mechanisms are required to ensure integrity, confidentiality and authenticity of the data.

WSNs face major security problems because the communication is wirelessly and the devices are often easy to access. Therefore, an adversary can easily eavesdrop on communication or simply steal a node. Since sensor nodes are usually not tamper-resistant, an adversary can often read out any content that

is stored on the node. Furthermore, the devices are very constrained in terms of memory, computing power, and energy supply. Since battery powered devices have a limited amount of energy, the major metric in the area of WSNs is energy consumption. The lifetime of a WSN is directly proportional to its energy efficiency, i.e., the less energy is consumed by applications the longer the batteries will last.

Symmetric algorithms are generally preferable to asymmetric algorithms in the field of WSNs because they are more efficient in terms of energy consumption and memory requirements. However, when symmetric algorithms are used, two problems arise: (1) key distribution and (2) number of stored keys. When individual keys are used in a WSN with n nodes, each node has to store $(n - 1)$ keys. This has good resiliency properties but obviously scales badly and is especially unsuitable for large WSNs. Moreover, perfect forward secrecy is not given after a node's key have been compromised. When one single symmetric key is used, memory requirement is greatly reduced, but at the same time this is not resilient anymore. To cope with this problem many probabilistic key distribution schemes for symmetric algorithms have been proposed [EG02, CPS03, DDHV]. In general these approaches either need pre-distributed keys, which means a higher configuration effort before deployment, or they produce much traffic, which results in higher energy consumption. Therefore, asymmetric algorithms are very valuable for key establishment and authentication in WSN.

Asymmetric cryptography has long seen as being too demanding for constrained devices such as sensor nodes with an 8-bit micro controller. However, there exist several protocols for asymmetric cryptographic algorithms for WSNs. In [WKC⁺04] Watro et al. describe public-key based protocols for WSNs. In particular, they present authentication and key-agreement protocols based on RSA. The so-called TinyPK was implemented in NesC for MicaZ 8-bit micro controller. However, one RSA exponentiation with a 1024-bit key needs 14.5 seconds, which is arguably not acceptable in many applications. RSA needs much longer key lengths compared to elliptic curve cryptography to achieve the same security level (1024 bit vs. 160-bit) [Res00]. Considering the limited amount of memory, computing power and energy of a typical 8-bit sensor node, it seems that ECC is a much better choice for public-key cryptography for WSN rather than RSA. Since TinyPK is based on the more demanding RSA algorithm and was implemented in NesC, it is not surprising that this is more than one order of magnitude slower than the fastest known implementation of a point multiplication for ECC in assembly. In [GPW⁺04] Gura et al. describe a point multiplication on a 160-bit standard curve within 0.81 seconds. The majority (77%) of the clock cycles was required by the modular multiplication. However, the source code of this implementation is not publicly available, it is rather intellectual property of Sun Microsystems. Therefore, these impressive results are not usable for the scientific community. Alternatively there is the TinyECC implementation [LN06], which may be used free of charge. TinyECC is a free software package for TinyOS that supports all SECG recommended 128-bit, 160-bit and 192-bit elliptic curve domain parameters. However, it is slower and needs more memory than the

equivalent of SUN Microsystems. Therefore, our goal was to implement a prime field arithmetic for an ECC scheme for 8-bit micro controller, which is open source and at the same time faster than the aforementioned implementation of SUN.

The remainder of this work is organized as follows: In Section 2 we give an introduction to elliptic curve cryptography and constraints of the target devices. Subsequently, in Section 3 our implementation of the modular multiplication for a 160-bit standard elliptic curve is described. The results of our implementation are presented in Section 4. Finally, this paper is concluded in Section 5.

2 Preliminary Assumptions and Introduction to Elliptic Curve Cryptography

In this section, we first state the constraints of the target micro controller. Subsequently we introduce the mathematical background of ECC. Finally, we state the implementation issues that arise when trying to implement ECC for constrained devices.

2.1 Constrained Devices

For the envisioned applications of WSNs, up to tens of thousands of smart, but battery powered devices are required, which communicate wirelessly. In order to lower costs, these devices will be very constrained in terms of memory capacity, computing power and energy supply. Nowadays, the de-facto standard sensor nodes for researchers are the so-called Mica motes [xbo, HSW⁺00]. They comprise an 8-bit RISC ATMEL AVR ATmega128L [Atm] micro controller, 4 KB configuration EEPROM memory, 512 KB data Flash memory, 128 KB program Flash memory, various sensors, ZigBee radio interface, and two standard AA batteries. Ideally these batteries should last for several months up to years. Therefore, a small power consumption is a crucial requirement for any application running on these nodes. Sending and receiving of messages is by far the most energy consuming task on the nodes [HMV04], therefore the traffic should be minimized wherever possible. Furthermore, the energy consumption of an application is mainly determined by its execution time. Therefore, a rule-of-thumb is: the shorter the processing time of an algorithm, the lower its energy consumption.

2.2 Introduction to Elliptic Curve Cryptography

Compared with symmetric algorithms the asymmetric algorithms work very slow. In particular on low-power processors they are felt as not practical and are used only rarely or not at all. For this purpose special algorithms were developed, but they have to be cryptanalyzed and shown to be secure, which takes a long time, before they are suitable for protecting sensitive data or application. Elliptic curves represent a special case. The advantage of the Elliptic Curve

Cryptography (ECC) is that on one hand it is meanwhile quite well investigated and thus considered secure while on the other hand just a very short bit length is needed as compared to other asymmetric systems. In order to reach a security level, which is equivalent to an RSA key with a length of 1024-Bit, already 160 bits are sufficient with elliptic curves [Res00]. This is a ratio of 6.4 and will significantly reduce the consumed energy for key establishment.

Let E be an elliptic curve defined over a field K as shown in figure 1, then a set of points can be created by a *chord-and-tangent rule* (extended addition). If P and Q are two different points, which are part of the set, that intersect the elliptic curve in a straight line, there will be a third intersection on the straight line with the curve. The reflection on the x axis of the latter is called R and represents the sum of P and Q . Doubling works the same, but the straight line is given by the tangent of the curve in the according point. This set of points defined by the extended addition extended by the point ∞ forms an Abelian group. $P + P$ is referred to as $2P$. Accordingly is $P + .. + P = kP$. For every point P exist a point Q with $P = kQ$, if P is not the identity and the order of the elliptic curve is prime. Finding the appropriate k for a given set (Q, P) is considered to be hard and called the *elliptic curve discrete logarithm problem* (ECDLP). Most ECC protocols rely on the ECDLP.

There are various algorithms for the extended addition on an elliptic curve for different coordinates and different underlying fields. They can be optimized according to the used protocol and hardware. A good overview is given by [HC02] and [Bro01]. Regardless which algorithm is used, they are all based on the arithmetic of the underlying field. Especially the multiplication in the field comes at great cost in time and energy. An efficient field arithmetic is therefore the base for an efficient implementation of an elliptic curve cryptographic system.

As prime fields are potential to be implemented in software with good performance, we rely in the following on elliptic curves of the form

$$E/K : y^2 = x^3 + ax + b, \text{char}(K) \neq 2, 3 \tag{1}$$

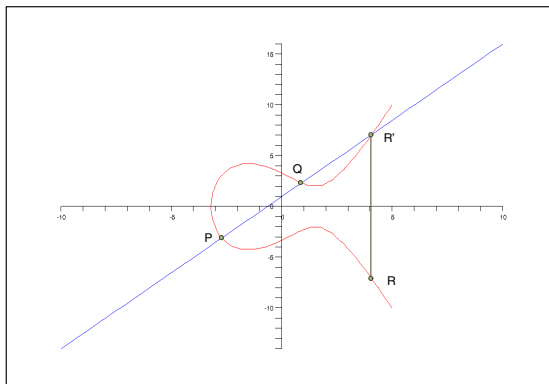


Fig. 1. Elliptic Curve, Parameters: a=-7 and b=11

2.3 Elliptic Curve Cryptography Implementation Issues

The basis for an efficient cryptographic system based on elliptic curves is a very efficient prime field arithmetic. As shown in Figure 2, a cryptographic system based on elliptic curves can be divided into three layers. The highest level actually represents the application layer. Protocols implemented here are for example ECDSA [HC02] or EC ElGamal [HC02]. Optimizations in this layer vary strongly, depending on the application (signature, coding etc.) and have to be partly or completely redone for each application. The underlying layer is the arithmetic of the elliptic curve. Most protocols are based on the multiplication of a point on the elliptic curve with an integer ($k * P$). However, optimizations at this level usually also strongly depend on the protocol layer. Optimizations in the underlying prime field arithmetics layer will always improve the performance of the whole ECC-System, because they are layer independent. More than 77% of the computing time can be applied here. Therefore, a very efficient prime field arithmetic is crucial for ECC based systems on constrained devices and time critical systems.

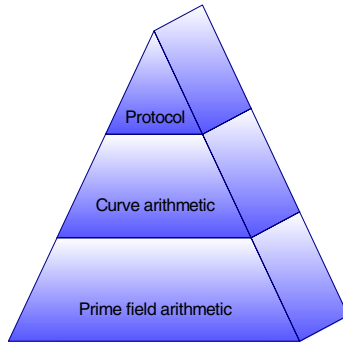


Fig. 2. Three Layers of an ECC-system

3 Implementation of Modular Multiplication

In this section, we first state criteria for an efficient implementation of an ECC system. Subsequently we will present details of our implementation of the modular multiplication, on which ECC system are based on.

3.1 Criteria for an Efficient ECC Implementation

Since optimizations in the prime fields arithmetic, contrary to other optimizations, will always improve the performance of the ECC system, the main attention goes here. Further optimization should be done depending on the application and the selected EC domain parameters. Prime field arithmetic should provide the operations multiply, add, subtract, halve and reduction. Operations with the most potential for optimization are the multiplication and the reduction. Starting point for the implementation is to choose a curve. For security reasons it

should be a standardized curve with at least 160 bit in length. To keep computations fast the bit length should be as short as possible. The curve “secp160r1” standardized by *Standards for Efficient Cryptography (SEC2)* [Cer00] was chosen for our implementation. It has two advantages that can be used to speed up prime field arithmetic reduction and to speed up curve arithmetic double and add. Because its underlying prime field is based on a pseudo Mersenne prime the reduction in the prime field can be done by several shifts and adds [Sol99] which is much faster than any other known algorithm on constrained devices. The curve parameter $a = -3$ can be used to reduce the effort of point doubling and point addition when using Jacobian projective coordinates [HC02].

To adapt the algorithms in the best possible way to the hardware the prime field arithmetic is completely implemented in assembly. As mentioned before the reduction can be implemented very efficiently if pseudo Mersenne primes are used. Addition and subtraction can be done without special optimization. The highest cost of computation lies in the 160-bit multiplication of the prime field. When choosing an algorithm for this multiplication it is important to consider the hardware characteristic, such as processor word-size and number of general purpose registers. The ATmega128L is able to perform an 8-bit multiplication in two cycles. Loading one 8-bit word from SRAM to registers also requires two cycles. Basically two different approaches are possible:

1. reduce the number of multiplication or
2. reduce SRAM usage.

The first attempt would be to implement Karatzuba [MVPV96] and the second some kind of *improved schoolbook* algorithm. The hybrid multiplication [GPW⁺04] is a memory optimized variant of the schoolbook algorithm. A special characteristic of the algorithm is that the computational cost rises linearly with smaller numbers of registers and processor word size. It also is much easier to implement than Karatzuba and hence much easier to port to different platforms. For these reasons the hybrid multiplication was chosen.

When doing a multiplication using the schoolbook algorithm the multiplication is divided in several parts that are accumulated to get the final result. The summands can be sorted in two ways before the addition: adding them from left-to-right or right-to-left¹ it is called *row wise multiplication*, see Figure 3(a). Sorting them by bit length is called *column wise multiplication*, see Figure 3(b). The hybrid multiplication algorithm [GPW⁺04] combines both methods: the summands that are used in the column wise way are calculated by using the row wise method, see Figure 4.

The number of rows per column is called *column width* (d). According to [GPW⁺04] the optimal column width is:

$$d = \max\{i \mid 1 \leq i \leq n, r \geq 3i + \lceil \log_2(n/i)/k \rceil\}, \quad (2)$$

where n is the operand size, r are the available registers and k is the bitlength.

¹ This is what is taught in school when learning the multiplication the first time - probably giving the algorithm its name.

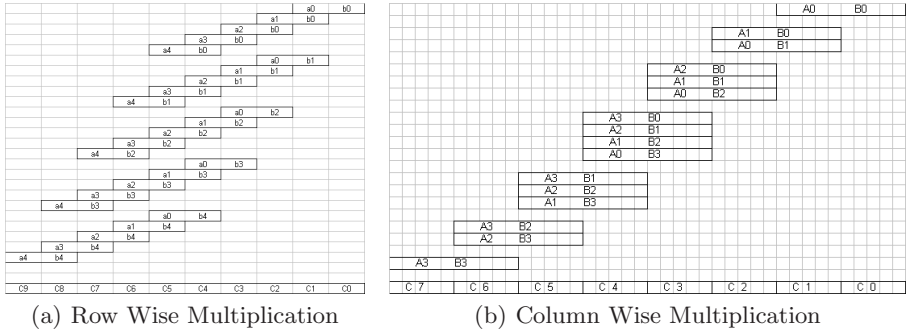


Fig. 3. Row Wise and Column Wise Multiplication

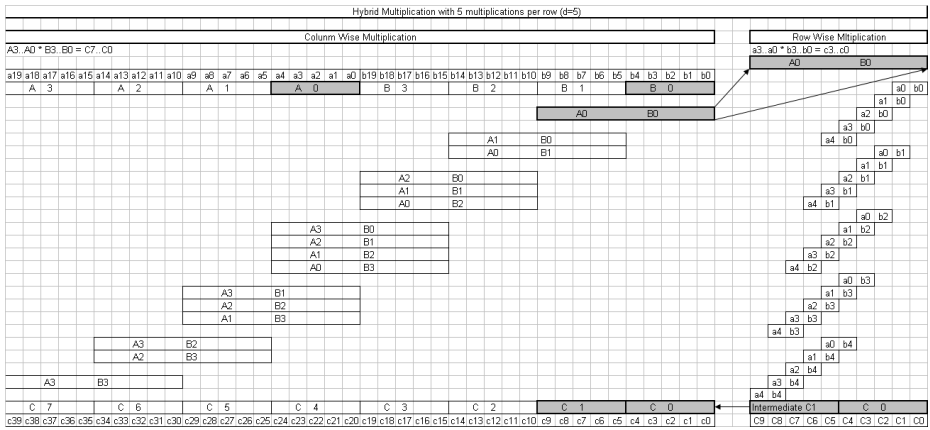


Fig. 4. 160-bit Hybrid Multiplication on ATmega128L with five Multiplications per Row

3.2 Implementation of the Modular Multiplication

According to Formula 2 the optimal d is 10 using all registers of the micro controller. In our first approach this parameter was used. The implementation benchmark showed that the implementation was about 50% slower than the benchmarks of SUN Microsystems in [GPW⁺04]. This overhead was mainly caused by handling carry bits. Let's have a look at the theoretic minimum effort of the algorithm. The core of the row wise part is the elemental 8 bit multiplication of the CPU followed by two additions to add the product to an intermediate result. These three operations are performed in the inner loop and will be referenced as the *elementary instruction block* in the remainder as illustrated in Figure 5(a). When using 160-bit operands this is done exact 400 times regardless of d . One multiplication and two additions equal 4 cycles. This means 1600 cycles in total plus the effort to get the operands from SRAM and write them

back. This effort depends on the parameter d which depends on the machine’s hardware. For the theoretic best d ($d = 10$) on our target device the memory load and store effort would be 80 data loads and 40 stores consuming 240 cycles in total. For $d = 5$ the data load effort would double to 160 cycles while data store effort remains at 40 consuming 400 cycles in total. In summary, the theoretic optimum is 1840 cycles for d equal to 10 or 2000 cycles for d equal to 5. However, our first implementation needed about 4500 cycles, even though we used the -theoretical- optimal column width d of size 10.

We found that surprisingly, the major part of the overhead was caused by carry handling rather than handling pointers or other arbitrary effort. The elementary instruction block is one 8-bit multiplication followed by two additions as mentioned before. Since the additions are targeted to an intermediate result which is in general not zero the addition produces a carry bit in the general case. When the next iteration starts the elementary 8-bit multiplication will overwrite the carry flag in the CPU. Hence the carry bit has to be stored and restored in each *elementary instruction block*, which would result, in at least two additional cycles per elementary instruction block or an overhead of at least 66.66% only for carry handling! Note that at the end of each row and also at the end of each column additional carry handling is required. Even if an efficient carry store and restore is available, the operation “add with carry” would add the carry to the wrong register, as can be seen in Figure 5(b). The best solution we found that solves both problems requires three additional cycles per iteration of each *elementary instruction block*. Compared to the four cycles of the *elementary instruction block*, this is an overhead of 75%. Any other possible solution found needed more spare registers.

In our second implementation the column width d was chosen equal to 5. Note that in this case a 160-bit multiplication consists of 16 columns, each of them is

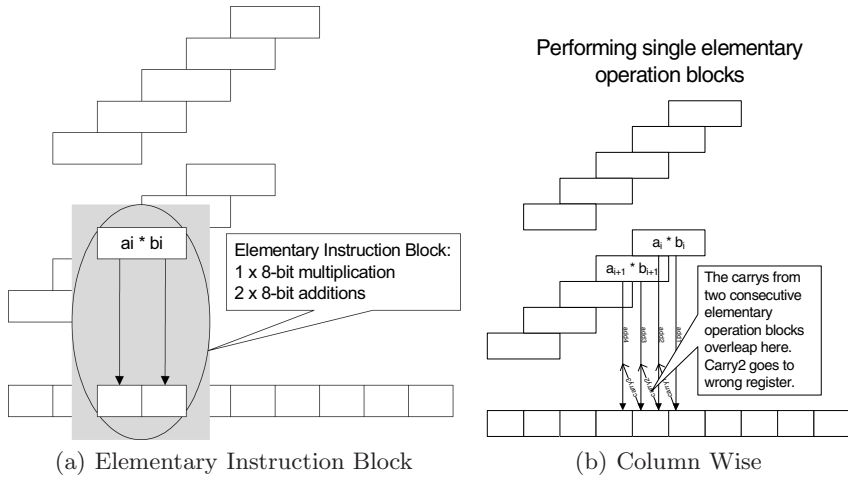


Fig. 5. Carry Handling Problems with *Elementary Instruction Blocks*

comprised of five rows. Five *elementary instruction blocks* are required to calculate one row. Furthermore, by halving the column width d the number of memory loads is doubled. In other words, we trade at least 80 additional cycles for the sake of more spare registers. Storing and restoring the carry bit after each 8-bit multiplication is not efficient. Several different solutions are possible, but discussing them all would exceed the frame of this work. A solution in which the carry bit can be handled by the “add with carry” command is required. In the next subsections we will emphasize the overhead produced by carry handling within one row and within one column. Finally we will summarize the carry handling costs.

Calculating a Row: The number of consecutive *elementary instruction blocks* performed in the row wise part is set by the parameter d . In this case five iterations are done in a row. The spare registers can be used as a buffer to safe the five 16-bit products of the five 8-bit multiplications, see Figure 6. After the five multiplications are executed and buffered, eleven additions follow, which are performed in the order shown by the numbers in Figure 6. Addition number six is represented by the -carryadd- arrow. It represents a normal “add with carry” instruction, that adds a zero to the register holding the high significant byte of the result of an 8-bit multiplication, thus adding the carry bit. We call this carry add “secure” because it cannot produce another carry. This is due to the fact that the maximal product $0xFF * 0xFF = 0xFE01$. Hence, adding a carry bit to the high significant byte of $0xFE01$ results in $0xFF01$ and does not produce another carry bit. This *serialization/pipelining of elementary instruction blocks* reduces the carry handling within a row to four move instructions (the last multiplication does not need to be buffered) and one addition instruction or, respectively, one clock cycle per *elementary instruction block*. Note that the previous approach required three cycles per *elementary instruction block* for carry handling. In other words the overhead is reduced from 75% in the first approach to now 25%. However, again additional handling is needed for carry bits occurring at the end of each row and column.

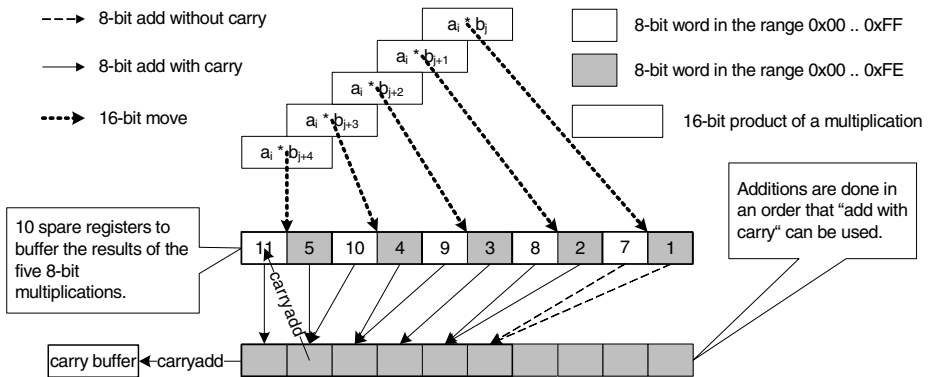


Fig. 6. Carry Handling in one Row

Calculating a Column: Recall that a column is comprised of five rows, i.e. five rows have to be processed to calculate a column. The last addition done in a row produces a carry bit which has to be processed in one of the upcoming rows, as we will see below. Figure 6 shows the carry handling within columns. A white box denotes an 8-bit register holding a value smaller than 0xFE, i.e. a “secure” carry add is possible with this register, whereas a gray box denotes an 8-bit register with an arbitrary value. As mentioned before, the carry bit which occurs at the end of each row needs to be processed later on, therefore it is buffered either in “carry buffer 1” or in “carry buffer 2”. The correct position where this carry bit has to be added is displayed by the position of the carry buffer holding it. Figure 6 shows furthermore, that in two successive rows the latter one has no register in which the carry bit of the former could be “securely” added. In the subsequent row this is possible, hence a second carry buffer is required. The two buffers are used alternating to save the carry bits, which occur after the calculation of each row. Therefore, two additional cycles overhead are required for carry handling for each column.

The carry bit occurring at the end of the column is stored in a third buffer. Since more than one row may be calculated using the same accumulator bytes, more than one carry bit is accumulated in the third carry buffer. If the next column starts with new accumulator bytes the carry buffer has to be processed. Figure 7 shows the correct position. In this case two additions are done, whereby the latter is “secure”. This is because the carry buffer may exceed the value 0x01 making a single “secure” carry add impossible. Therefore, three additional cycles for carry handling are required if columns start with a new accumulator.

Summary of Carry Handling Costs: This way the total carry handling results in:

- 5 cycles for 5 elementary instruction blocks (equals 1 row)
- 2 cycles for each column
- 3 cycles for each column starting with new accumulator

Altogether $\frac{400+32+15}{400} = 1.1175$ additional cycles² per *elementary instruction block* are required for the carry handling, which is equivalent to an overhead of 28%. Note that this calculation includes all carry handling for the entire multiplication, whereas in the estimation of our first approach (75% overhead) additional carry handling at the end of each row and column was required. Since the *elementary instruction block* is repeated 400 times the benefits in saving both time and energy is enormous.

Two more aspects shall be mentioned here: First, the amount of needed registers to apply this carry handling equals the number of partial product which have to be buffered per row. As a result a smaller d has to be applied. Choosing the optimum size for d in reality can be a quite challenging task though.

² Recall that each column is comprised of 5 rows, each of them is comprised of 5 *elementary instruction blocks*, i.e. each column consists of 25 *elementary instruction blocks*. For a 160-bit point multiplication 16 columns are required, i.e. 400 *elementary instruction blocks*. Five columns starting with a new accumulator require additional cycles.

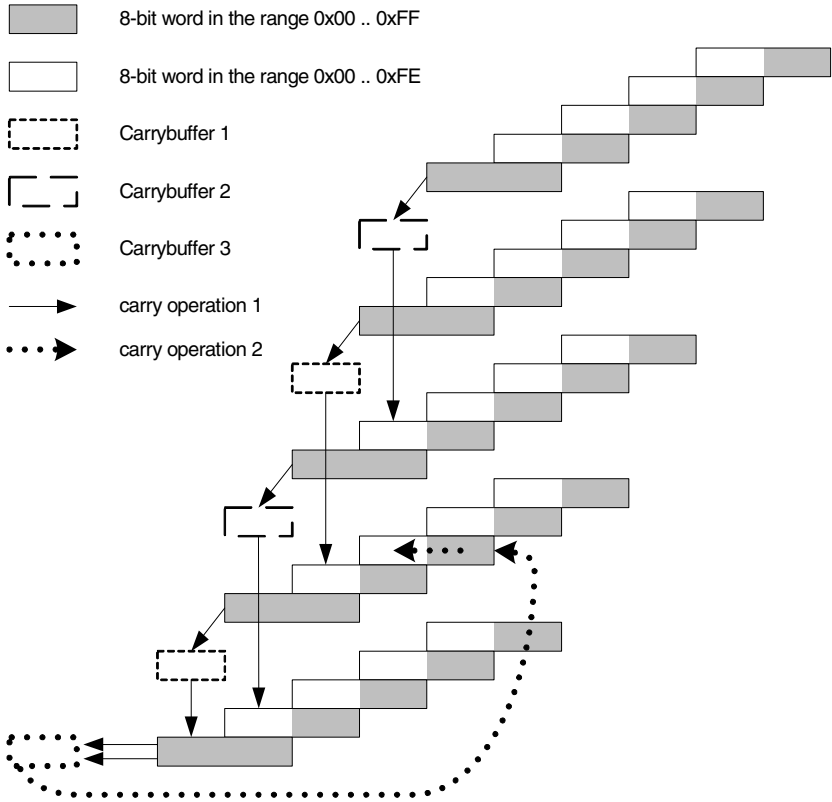


Fig. 7. Carry Handling in Columns

Second, the additional effort for handling carry bits in the way presented here can be divided in a static and a dynamic part. The effort of one clock cycle per *elementary instruction block* is static, while the remainder is supposed to grow with smaller column width.

4 Results

The basic requirement for a fast and thus energy efficient implementation of ECC is a very fast multiplication in the prime field. The fastest known implementation was implemented by SUN Microsystems. In [GPW⁺04] they provide a benchmark for the micro controller that we used as well, hence a direct comparison is possible. A 160-bit multiplication from SUN Microsystems' implementation requires 3106 cycles, which is at a clock rate of 7.37 MHz equivalent to 0.42 ms.

The implementation presented in this work needs 2881 cycles for a 160-bit multiplication, which is equivalent to 0.39 ms at 7.37 MHz. In fact, this represents a time saving of 7.2%. To the best of our knowledge this is the fastest

Table 1. Overview of instructions used

Instr.	#C/I	This work		SUN Microsystems		Theoretical Min.	
		Instr.	Cycles	Instr.	Cycles	Instr.	Cycles
add/adc	1	986	986	1360	1360	800	800
mul	2	400	800	400	800	400	800
ld/lds	2	238	476	167	334	160	320
st/sts	2	40	80	40	80	40	80
mov/movw	1	355	355	335	335		
other			184		197		
Sum			2881		3106		2000

implementation world wide of a modular multiplication of a 160-bit standardized elliptic curve for an 8-bit micro controller.

In Table 1 we present a detailed list of instructions used by our and SUN Microsystems' implementation as published in [GPW⁺04]. A third column contains the theoretical minimum amount of the appropriate instruction, as required by the hybrid multiplication with a column width of 5 on the ATMega128L micro controller. However, this number cannot be achieved, but is mentioned to show the limit and the overhead. Each row represents an instruction or a set of instructions, which are very similar. The first row represents the 8-bit addition with and without carry. In the next row the number of 8-bit multiplications can be seen. In the following row all used data loads are combined. Thereafter the used commands to write back to SRAM are listed. The underlying row shows all 8-bit and 16-bit register moves. Finally all other instructions are combined. In this row only the number of used cycles is given while the number of instructions is missing, because different instructions may consume different number of cycles to be executed.

As one can see, the main differences between our implementation and SUN Microsystems' lie in the number of used additions and data loads. Note that data loads require two cycles contrary to the addition, which only requires one cycle. Although SUN Microsystems' implementation executes less data load instructions, in total it requires more cycles than our implementation. The time saving results from the improved carry handling reducing the number of needed additions close to the minimum. In SUN Microsystems' implementation the number of data loads is close to the minimum number of 160 data loads for a column width of 5. The additional data loads in our implementation result from pointer handling. Pointers have to be restored from SRAM very often, because the carry handling needs all spare registers.

Comparison with TinyECC is cumbersome for two reasons: on the one hand neither time tables for curve nor modular arithmetic for TinyECC are available. On the other hand we did not implement a full ECDSA protocol. Therefore we estimate the execution time of an ECDSA signature based on our modular multiplication. [GPW⁺04] state that 77% of the execution time of one point multiplication are required for modular multiplication. Assuming our multiplication to be used here would result in 0.76s. Note that this curve arithmetic includes

some well applied algorithmic optimizations which are best fitted to hardware, because they are done in assembly. On the other hand no special optimization for ECDSA were included, e.g. the y-coordinate is calculated but not used at all for the ECDSA protocol. A signature requires one inversion, two modular multiplication, and one modular addition. In addition one SHA-1 has to be executed to hash the message. Generally SHA-1 and a modular multiplication are both roughly three orders of magnitude faster than a point multiplication. The execution time of an inversion is in the range of several modular multiplications. The execution time of the modular addition is roughly four orders of magnitude faster than the execution of a point multiplication. Therefore, we estimate that all required operations for an ECDSA signature, including the SHA-1, can most probably be performed in less than one second. A TinyECC ECDSA signature generation takes slightly less than two seconds, including the time for the SHA-1 execution. Furthermore, once a precomputation time of a 3.5s is required.

5 Conclusion and Future Work

We presented the fastest implementation of a modular multiplication for a 160-bit standardized elliptic curve for 8-bit micro controller in Section 3 and compared the results in Section 4. We also highlighted the criteria for efficient implementations of ECC schemes for 8-bit micro controller and pointed out the problems that arise when implementing

Since modular multiplications take up the major part of the computing time of point multiplications over an elliptic curve, our results can be used to significantly increase the efficiency of point multiplications over an elliptic curve. Many ECC schemes such as EC ElGamal or ECDSA are based on modular multiplication and will therefore directly benefit from our results. Our results bring the vision of asymmetric cryptography in the field of WSNs with all its benefits for key-distribution and authentication a step closer to reality.

Next steps are the efficient implementation of point multiplication over the elliptic curve and some ECC schemes such as EC ElGamal and ECDSA. Furthermore an integration into existing ECC modules for TinyOS is thinkable.

Acknowledgments

The authors would like to thank Bodo Möller and André Weimerskirch for their insights and comments on various aspects of this work. The work presented in this paper was supported in part by the European Commission within the STREP UbiSec&Sens of the EU Framework Programme 6 for Research and Development (www.ist-ubiseconsens.org). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the UbiSec&Sens project or the European Commission.

References

- [Atm] Atmel. 8-bit Microcontroller with 128K Bytes In-System Programmable Flash <http://www.atmel.com/>
- [Bro01] Brown, M., Hankerson, D., López, J., Menezes, A.: Software Implementation of the NIST Elliptic Curves Over Prime Fields. In: Naccache, D. (ed.) *Topics in Cryptology - CT-RSA 2001*. LNCS, vol. 2020, p. 250. Springer, Heidelberg (2001)
- [Cer00] Certicom Research. SEC 2: Recommended Elliptic Curve Domain Parameters. *Standards for Efficient Cryptography Version 1.0* (September 2000)
- [CPS03] Chan, H., Perrig, A., Song, D.: Random Key Predistribution Schemes for Sensor Networks. In: *Proceedings of the IEEE Security and Privacy Symposium 2003* (2003)
- [DDHV] Du, W., Deng, J., Han, Y., Varshney, P.: A Pairwise Key Pre-distribution Scheme for Wireless Sensor Networks. In: *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications Security*
- [EG02] Eschenauer, L., Gligor, V.: A Key Management Scheme for Distributed Sensor Networks. In: *CCS '02: Proceedings of the 9th ACM Conference on Computer and Communications Security*, ACM Press, New York (2002)
- [GPW⁺04] Gura, N., Patel, A., Wander, A., Eberle, H., Shantz, S.C.: Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. In: Joye, M., Quisquater, J.-J. (eds.) *CHES 2004*. LNCS, vol. 3156, pp. 119–132. Springer, Heidelberg (2004)
- [HC02] Hankerson, D., Menezes, A.J., Vanstone, S.: *Guide to Elliptic Curve Cryptography*. Springer, Heidelberg (2004)
- [HMOV04] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System Architecture Directions for Networked Sensors. *SIGOPS Oper. Syst. Rev.* 34(5), 93–104 (2000)
- [HSW⁺00] Hill, J.L., Culler, D.: Mica: a Wireless Platform for Deeply Embedded Networks. *Micro, IEEE* 22(6), 12–24 (2002)
- [LN06] Liu, A., Ning, P.: TinyECC: Elliptic Curve Cryptography for Sensor Networks. available for download at (September 2006) <http://discovery.csc.ncsu.edu/software/TinyECC>
- [MVPV96] Menezes, A.J., Van, O., Paul, C., Vanstone, S.A. (eds.): *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FL (1996)
- [Res00] Certicom Research. SEC 1: Elliptic Curve Cryptography, Version 1.0 (September 2000)
- [Sol99] Solinas, J.: Generalized Mersenne Numbers. Technical report CORR-39, Dept. of C&O, University of Waterloo (1999) Available from <http://www.cacr.math.uwaterloo.ca>
- [WKC⁺04] Watro, R., Kong, D., Cuti, S.F., Gardiner, C., Lynn, C., Kruus, P.: TinyPK: Securing Sensor Networks with Public Key Technology. In: *SASN '04: Proceedings of the 2nd ACM Workshop on Security of Ad Hoc and Sensor Networks*, pp. 59–64. ACM Press, New York (2004)
- [xbo] Crossbow Technology, Inc., <http://www.xbow.com>