Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs

Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz

Sun Microsystems Laboratories
{Nils.Gura, Arun.Patel,Arvinderpal.Wander,Hans.Eberle, Sheueling.Chang}@sun.com
http://www.research.sun.com/projects/crypto

Abstract. Strong public-key cryptography is often considered to be too computationally expensive for small devices if not accelerated by cryptographic hardware. We revisited this statement and implemented elliptic curve point multiplication for 160-bit, 192-bit, and 224-bit NIST/SECG curves over GF(p) and RSA-1024 and RSA-2048 on two 8-bit microcontrollers. To accelerate multiple-precision multiplication, we propose a new algorithm to reduce the number of memory accesses.
Implementation and analysis led to three observations: 1. Public-key cryptography is viable on small devices without hardware acceleration. On an Atmel ATmega128 at 8 MHz we measured 0.81s for 160-bit ECC point multiplication and 0.43s for a RSA-1024 operation with exponent $e = 2^{16} + 1$. 2. The relative performance advantage of ECC point multiplication over RSA modular exponentiation increases with the decrease in processor word size and the increase in key size. 3. Elliptic curves over fields using pseudo-Mersenne primes as standardized by NIST and SECG allow for high performance implementations and show no performance disadvantage over optimal extension fields or prime fields selected specifically for a particular processor architecture.

Keywords: Elliptic Curve Cryptography, RSA, modular multiplication, sensor networks.

1 Introduction

As the Internet expands, it will encompass not only server and desktop systems, but also large numbers of small devices ranging from PDAs and cell phones to appliances and networked sensors. Inexpensive radio transceivers, integrated or attached to small processors, will provide the basis for small devices that can exchange information both locally with peers and, through gateway devices, globally with entities on the Internet. Deploying these devices in accessible environments exposes them to potential attackers that could tamper with them, eavesdrop communications, alter transmitted data, or attach unauthorized devices to the network. These risks can be mitigated by employing strong cryptography to ensure authentication, authorization, data confidentiality, and data integrity. Symmetric cryptography, which is computationally inexpensive, can be used to achieve some of these goals. However, it is inflexible with respect to key management as it requires pre-distribution of keys. On the other hand, public-key cryptography allows for flexible key management, but requires a significant amount of computation. However, the compute capabilities of low-cost CPUs are very limited in terms of clock frequency, memory size, and power constraints.

Compared to RSA, the prevalent public-key scheme of the Internet today, Elliptic Curve Cryptography (ECC) offers smaller key sizes, faster computation, as well as memory, energy and bandwidth savings and is thus better suited for small devices. While RSA and ECC can be accelerated with dedicated cryptographic coprocessors such as those used in smart cards, coprocessors require additional hardware adding to the size and complexity of the devices. Therefore, they may not be desirable for low-cost implementations. Only few publications have considered public-key cryptography on small devices without coprocessors. Hasegawa et al. implemented ECDSA signature generation and verification on a 10MHz M16C microcomputer [11]. The implementation requires 4KB of code space and uses a 160-bit field prime $p = 65112 * 2^{144} - 1$ chosen to accommodate the 16-bit processor architecture. Signatures can be generated in 150ms and verified in 630ms. Based on the ECC integer library, the authors also estimate 10s for RSA-1024 signature generation and 400ms for verification using $e = 2^{16} + 1$. Bailey and Paar suggest the use of optimal extension fields (OEFs) that enable efficient reduction when subfield primes are chosen as pseudo-Mersenne primes close to the word size of the targeted processor [2]. An implementation of this concept for elliptic curve point multiplication over $GF((2^8 - 17)^{17})$ on an 8-bit 8051 processor architecture is described by Woodbury, Bailey and Paar in [17]. On a 12MHz 8051 with 12 clock cycles per instruction cycle, the authors measured 8.37s for general point multiplication using the binary method and 1.83s for point multiplication with a fixed base point. The code size was 13KB and 183 bytes of internal and 340 bytes of external RAM were used. Pietiläinen evaluated the relative performance of RSA and ECC on smart cards [16].

This paper focuses on implementation aspects of standardized RSA and ECC over NIST/SECG $GF(p)$ curves and evaluates the algorithms with respect to performance, code size, and memory usage. We consider software and hardware optimization techniques for RSA and ECC based on implementations on two exemplary 8-bit microcontroller platforms: The 8051-based Chipcon CC1010 [6], and the Atmel AVR ATmega128 [1].

2 Public-Key Algorithms ECC and RSA

ECC and RSA are mature public-key algorithms that have been researched by the academic community for many years; RSA was conceived by Rivest, Shamir and Adleman in 1976 and Koblitz and Miller independently published work on ECC in 1985. The fundamental operation underlying RSA is modular exponentiation in integer rings and its security stems from the difficulty of factoring large integers. ECC operates on groups of points over elliptic curves and derives its security from the hardness of the elliptic curve discrete logarithm problem (ECDLP). While sub-exponential algorithms can solve the integer factorization problem, only exponential algorithms are known for the ECDLP. This allows ECC to achieve the same level of security with smaller key sizes and higher computational efficiency; ECC-160 provides comparable security to RSA-1024 and ECC-224 provides comparable security to RSA-2048.

2.1 Implementing RSA

RSA operations are modular exponentiations of large integers with a typical size of 512 to 2048 bits. RSA encryption generates a ciphertext C from a message M based on a modular exponentiation $C = M^e \mod n$. Decryption regenerates the message by computing $M = C^d \mod n$ [1]. Among the several techniques that can be used to accelerate RSA [3], we specifically focused on those applicable under the constraints of 8-bit devices.

Chinese Remainder Theorem RSA private-key operations, namely decryption and signature generation, can be accelerated using the Chinese Remainder Theorem (CRT). RSA chooses the modulus n as the product of two primes p and q, where p and q are on the order of $\sqrt{n}$ (e.g. for a 1024-bit n, p and q are on average 512 bits long). Using the CRT, a modular exponentiation for decryption $M = C^d \mod n$ can be decomposed into two modular exponentiations $M_1 = C_1^{d_1} \mod p$ and $M_2 = C_2^{d_2} \mod q$, where C_1, d_1, C_2 and d_2 are roughly half the size of n. Assuming schoolbook multiplication with operands of size $\frac{m}{2} = \frac{\lceil log_2(n) \rceil}{2}$, modular multiplications can be computed in roughly $\frac{1}{4}$ of the time as m-bit modular multiplications. Thus the CRT reduces computation time by nearly $\frac{3}{4}$ resulting in up to a 4x speedup.

Montgomery Multiplication An efficient reduction scheme for arbitrary moduli n is Montgomery reduction which computes $C' * r^{-1} \mod n$ instead of $C' \mod n$. Appendix B shows a simple algorithm for Montgomery reduction of a $2m$-bit integer C' to an m-bit integer C on a processor with a word size of k bits. Using schoolbook multiplication, an $m \times m$-bit multiplication requires $\lceil \frac{m}{k} \rceil^2$ $k \times k$-bit multiplications and Montgomery reduction requires $\lceil \frac{m}{k} \rceil^2 + \lceil \frac{m}{k} \rceil$ $k \times k$-bit multiplications. Therefore, the total cost of an $m \times m$-bit Montgomery multiplication is $2\lceil \frac{m}{k} \rceil^2 + \lceil \frac{m}{k} \rceil$.

Optimized Squaring The squaring of a large integer A, decomposed into multiple k-bit words $(A_{n-1}, ..., A_0)$, can take advantage of the fact that partial products $A_iA_j, i \neq j$ occur twice. For example, a squaring of $A = (A_1, A_0)$ needs to compute the partial product A_1A_0 only once since $A^2 = (A_1 * 2^k + A_0) = A_1A_1 * 2^{2k} + 2A_1A_0 * 2^k + A_0A_0$. Thus $m \times m$-bit squarings including

[1] A detailed description and a proof of mathematical correctness can be found e.g. in [3].

Montgomery reduction require only $\frac{3}{2}\lceil\frac{m}{k}\rceil^2 + \frac{3}{2}\lceil\frac{m}{k}\rceil$ $k \times k$-bit multiplications, reducing computational complexity by up to 25%.

2.2 Implementing ECC

The fundamental operation underlying ECC is *point multiplication*, which is defined over finite field operations[2]. All standardized elliptic curves are defined over either prime integer fields $GF(p)$ or binary polynomial fields $GF(2^m)$. In this paper we consider only prime integer fields since binary polynomial field arithmetic, specifically multiplication, is insufficiently supported by current microprocessors and would thus lead to lower performance. The point multiplication kP of an integer k and a point P on an elliptic curve $C : y^2 = x^3 + ax + b$ over a prime field $GF(p)$ with curve parameters $a, b \in GF(p)$ can be decomposed into a sequence of *point additions* and *point doublings*. Numerous techniques have been proposed to accelerate ECC point multiplication. Since we do not make assumptions about the point P in a point multiplication kP, optimization methods for fixed points cannot be applied. In the following, we will describe the most important optimization techniques for general point multiplication on elliptic curves over $GF(p)$ standardized by either NIST or SECG.

Projective Coordinate Systems Cohen et al. found that mixed coordinate systems using a combination of modified Jacobian and affine coordinates offer the best performance [7]. A point addition of one point in modified Jacobian coordinates $P_1 = (X_1, Y_1, Z_1, aZ_1^4)$ and one point in affine coordinates $P_2 = (x_2, y_2)$ resulting in $P_3 = (X_3, Y_3, Z_3, aZ_3^4)$ is shown in Formula 1 and a point doubling of a point in modified Jacobian coordinates P_1 is shown in Formula 2.

$$\begin{aligned} &X_3 = -H^3 - 2X_1H^2 + r^2, Y_3 = -Y_1H^3 + r(X_1H^2 - X_3), Z_3 = Z_1H \\ &aZ_3^4 = aZ_3^4 \quad \text{with} \quad H = x_2Z_1^2 - X_1, r = y_2Z_1^3 - Y_1 \end{aligned} \tag{1}$$

$$\begin{aligned} X_3 = T, Y_3 = M(S - T) - U, Z_3 = 2Y_1Z_1, aZ_3^4 = 2U(aZ_1^4) \\ \text{with} \quad S = 4X_1Y_1^2, U = 8Y_1^4, M = 3X_1^2 + (aZ_1^4), T = -2S + M^2 \end{aligned} \tag{2}$$

Using the above formulas, point addition requires 9 multiplications and 5 squarings and point doubling requires 4 multiplications and 4 squarings as the most expensive operations.

Non-Adjacent Forms Non-adjacent forms (NAFs) are a method of recoding the scalar k in a point multiplication kP in order to reduce the number of non-zero bits and thus the number of point additions [14]. This is accomplished by using digits that can be either 0, 1 or -1. For example, $15P = (1\,1\,1\,1)_2P$ can be represented as $15P = (1\,0\,0\,0\,\text{-}1)_2P$. The NAF of a scalar k has the properties

[2] For a detailed introduction to ECC the reader is referred to [10].

of having the lowest Hamming weight of any signed representation of k, being unique, and at most one bit longer than k. By definition, non-zero digits can never be adjacent resulting in a reduction of point additions from $\frac{m-1}{2}$ for the binary method to $\frac{m}{3}$ for NAF-encoded scalars. Negative digits translate into point subtractions, which require the same effort as point additions since the inverse of an affine point $P = (x, y)$ is simply $-P = (x, -y)$.

Curve-Specific Optimizations NIST and SECG specified a set of elliptic curves with verified security properties that allow for significant performance optimizations [15] [4]. For all NIST and most SECG curves, the underlying field primes p were chosen as pseudo-Mersenne primes to allow for optimized modular reduction. They can be represented as $p = 2^m - \omega$ where ω is the sum of a few powers of two and $\omega \ll 2^m$. Reduction of a $2m$-bit multiplication result C' split into two m-bit halves c'_1 and c'_0 can be computed based on the congruence $2^m \equiv \omega$:

$$
\begin{aligned}
&C' = (c'_1, c'_0) = A * B \\
&\texttt{while } (c'_1 \neq 0) \\
&\quad (c'_1, c'_0) = c'_1 * \omega + c'_0 \\
&C = c'_0 \mod p
\end{aligned}
$$

Since pseudo-Mersenne primes are sparse, multiplication by ω is commonly implemented with additions and shifts. It is important to note that compared to Montgomery reduction, reduction for pseudo-Mersenne primes requires substantially less effort on devices with small processor word sizes k. As described in Section 2.1, Montgomery reduction of a $2m$-bit product to an m-bit result requires $\lceil \frac{m}{k} \rceil^2 + \lceil \frac{m}{k} \rceil$ $k \times k$-bit multiplications. The ratio $\lceil \frac{m}{k} \rceil^2$ grows by the square as the processor word size k is decreased. Reduction for NIST/SECG pseudo-Mersenne primes, however, typically only requires two multiplications with a sparse ω. The number of corresponding additions and shifts scales linearly with the decrease of k. For example, if n multiplications were needed for Montgomery reduction on a 32-bit processor, $16n$ multiplications would be needed on an 8-bit processor. On the other hand, if a additions were needed for pseudo-Mersenne prime reduction on a machine with a 32-bit processor, only $4a$ additions would be needed on an 8-bit processor. Other ECC operations such as addition and subtraction also scale linearly. As a result, implementations of ECC exhibit a relative performance advantage over RSA on processors with small word sizes. Assuming a constant number of addends in the pseudo-Mersenne field primes, the advantage of ECC over RSA on devices with small word sizes likewise grows with the key size.

All NIST and some SECG curves further allow for optimization based on the curve parameter a being $a = -3$. Referring to point doubling Formula 2, M can be computed as $M = 3X_1^2 - 3Z_1^4 = 3(X_1 - Z_1^2) * (X_1 + Z_1^2)$ and aZ_3^4 as $aZ_3^4 = 6UZ_1^4$. As a result, aZ_3^4 does not have to be computed in point addition Formula 1 such that point doublings can be performed with 4 multiplications and 4 squarings and point additions can be performed with 8 multiplications and 3 squarings. Similar optimizations were used by Hitchcock et al. [12] and Hasegawa et al. [11].

3 Optimizing Multiplication for Memory Operations

Modular multiplication and squaring of large integers are the single performance-critical operations for RSA and ECC as we will show in Section 4. Therefore, high-performance implementations need to focus specifically on optimizing these operations. On small processors, multiple-precision multiplication of large integers not only involves arithmetic operations, but also a significant amount of data transport to and from memory due to limited register space. To reduce computational complexity, we considered Karatsuba Ofman [13] and FFT multiplication, but found that the recursive nature of these algorithms leads to increased memory consumption and frequent memory accesses to intermediate results and stack structures. In addition, Karatsuba Ofman and FFT multiplication cannot be applied to Montgomery reduction due to dependencies of the partial products. We therefore decided to focus on optimizing schoolbook multiplication. For schoolbook multiplication of m-bit integers on a device with a word size of k bits, the multiplication effort for m-bit integers is fixed to $n^2 = \lceil \frac{m}{k} \rceil^2$ $k \times k$-bit multiplication operations plus appendant additions.

Therefore, computation time can mainly be optimized by reducing the number of non-arithmetic operations, specifically memory operations. Table 1 illustrates and analyzes three multiplication strategies with respect to register usage and memory operations. It shows exemplary multiplications of n-word integers $(a_{n-1}, \ldots, a_1, a_0)$ and $(b_{n-1}, \ldots, b_1, b_0)$. The analysis assumes that multiplicand, multiplier and result cannot fit into register space at the same time such that memory accesses are necessary.

3.1 Row-Wise Multiplication

The row-wise multiplication strategy keeps the multiplier b_i constant and multiplies it with the entire multiple-precision multiplicand $(a_{n-1}, \ldots, a_1, a_0)$ before moving to the next multiplier b_{i+1}. Partial products are summed up in an accumulator consisting of n registers $(r_{n-1}, \ldots, r_1, r_0)$. Once a row is completed, the last register of the accumulator (r_0 for the first row) can be stored to memory as part of the final result and can be reused for accumulation of the next row. Two registers are required to store the constant b_i and one variable a_j. In the above implementation, row-wise implementation requires $n + 2$ registers and performs $n^2 + 3n$ memory accesses[3]. That is, for each $k \times k$ multiplication one memory load operation is needed. On processor architectures that do not have sufficient register space for the accumulator, up to $n^2 + 1$ additional load and $n^2 - n$ additional store operations are required. On the other hand, processors that can hold

[3] Additional registers may be required for pointers, multiplication results and temporary data storage. We do not consider them in the analysis since they depend on the processor architecture.

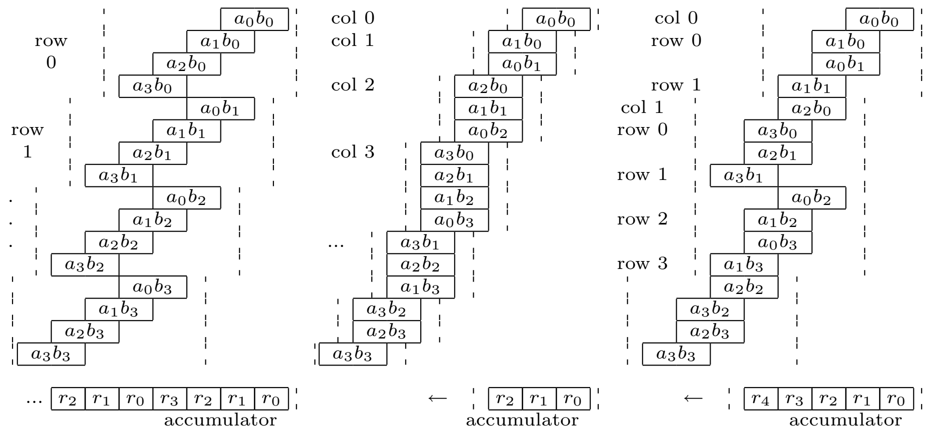
both the accumulator and the entire multiplicand in register space can perform row-wise multiplication with $2n + 1$ registers and only $4n$ memory accesses. In addition to memory accesses, pointers to multiplicand, multiplier and result may have to be adjusted on implementations using indexed addressing. If multiplicand and multiplier are indexed, one pointer increment/decrement is needed for each load operation, which is true for all three multiplication algorithms.

3.2 Column-Wise Multiplication

The column-wise multiplication strategy sums up columns of partial products $a_j * b_i$, where $i + j = l$ for column l. At the end of each column, one k-bit word is stored as part of the final multiplication result. Column-wise multiplication requires $4 + \lceil log_2(n)/k \rceil$ registers, the fewest number of all three algorithms. It is interesting to note that the number of registers grows only negligibly with the increase of the operand size n. Column-wise multiplication is thus well suited for architectures with limited register space. However, $2n^2 + 2n$ memory operations have to be performed, which corresponds to two memory load operations per $k \times k$ multiplication. Implementations of column-wise multiplication require advancing pointers to both multiplicand a_j and multiplier b_i once for every $k \times k$-bit multiplication.

3.3 Hybrid Multiplication

We propose a new hybrid multiplication strategy that combines the advantages of row-wise and column-wise multiplication. Hybrid multiplication aims at optimizing for both the number of registers and the number of memory accesses. We employ the column-wise strategy as the "outer algorithm" and the row-wise strategy as the "inner algorithm". That is, hybrid multiplication computes columns that consist of rows of partial products. The savings in memory bandwidth stem from the fact that k-bit operands of the multiplier are used in several multiplications, but are loaded from memory only once. Looking at column 0 in the example, b_0 and b_1 are used in two multiplications, but have to be loaded only once. Register usage and memory accesses depend on the the number of partial products per row (or column width) d. The hybrid method equals the column-wise strategy for $d = 1$ and it equals the row-wise strategy for $d = n$, where the entire multiplicand is kept in registers. d can be chosen according to the targeted processor; larger values of d require fewer memory operations, but more registers to store operands and to accumulate the result. To optimize the algorithm performance for r available registers, d should be chosen such that $d = max\{i|1 \leq i \leq n, r \geq 3i + \lceil log_2(n/i)/k \rceil\}$. Note that the number of registers grows only logarithmically with the increase in operand size n. Therefore, for a fixed value of d, hybrid multiplication scales to a wide range of operand sizes n without requiring additional registers. This is important for implementations that have to support algorithms such as RSA and ECC for multiple key sizes. The hybrid multiplication algorithm is shown in pseudo code in Appendix A.



	Row-Wise Multiplication	Column-Wise Multiplication	Hybrid Multiplication ($d = 2$)
accumulator registers	n	$2 + \lceil log_2(n)/k \rceil$	$2d + \lceil log_2(n/d)/k \rceil$
operand registers	2	2	d+1
memory loads	$n^2 + n$	$2n^2$	$2\lceil n^2/d \rceil$
memory stores	$2n$	$2n$	$2n$
registers	$n + 2$	$4 + \lceil log_2(n)/k \rceil$	$3d + \lceil log_2(n/d)/k \rceil$
memory ops	$n^2 + 3n$	$2n^2 + 2n$	$2\lceil n^2/d \rceil + 2n$

Table 1. *Multiple-precision multiplication of integers with $n = 4$ words.*

Algorithm	**ATmega128 @ 8MHz**			**CC1010 @ 14.7456MHz**		
	time	data mem	code	time	data mem	code
	s	bytes	bytes	s	ext+int, bytes	bytes
ECC secp160r1	0.81s	282	3682	4.58s	180+86	2166
ECC secp192r1	1.24s	336	3979	7.56s	216+102	2152
ECC secp224r1	2.19s	422	4812	11.98s	259+114	2214
Mod. exp. 512	5.37s	328	1071	53.33s	321+71	764
RSA-1024 public-key $e = 2^{16} + 1$	0.43s	542	1073	> 4.48s		
RSA-1024 private-key w. CRT	10.99s	930	6292	~ 106.66s		
RSA-2048 public-key $e = 2^{16} + 1$	1.94s	1332	2854			
RSA-2048 private-key w. CRT	83.26s	1853	7736			

Table 2. *Average ECC and RSA execution times on the ATmega128 and the CC1010.*

The execution time for the RSA-1024 private-key operation on the CC1010 was approximated as twice the execution time of a 512-bit Montgomery exponentiation and the execution time for the RSA-1024 public-key operation was estimated as four times the execution time of a 512-bit Montgomery exponentiation using $e = 2^{16}+1$. Since only one 512-bit operand and no full 1024-bit operand can be kept in internal memory, an actual implementation of the RSA-1024 public-key operation would be even less efficient.

4 Implementation and Evaluation

We implemented ECC point multiplication and modular exponentiation on two exemplary 8-bit platforms in assembly code. As the first processor, we chose a Chipcon CC1010 8-bit microcontroller which implements the Intel 8051 instruction set. The CC1010 contains 32KB of FLASH program memory, 2KB of external data memory and 128 bytes of internal data memory. As part of the 8051 architecture, 32 bytes of the internal memory are used to form 4 banks of 8 8-bit registers for temporary data storage. One 8-bit accumulator is the destination register of all arithmetic operations. The CC1010 is clocked at 14.7456MHz with one instruction cycle corresponding to 4 clock cycles such that the clock frequency adjusted for instruction cycles is 3.6864MHz.

As the second processor, we chose an Atmel ATmega128, a popular processor used for sensor network research, for example on the Crossbow motes platform [8]. The ATmega128 is an 8-bit microcontroller based on the AVR architecture and contains 128KB of FLASH program memory and 4KB of data memory. Unlike the CC1010, the ATmega128 implements a homogeneous data memory that can be addressed by three 16-bit pointer registers with pre-decrement and post-increment functionality. The register set consists of 32 8-bit registers, where all registers can be destinations of arithmetic operations. The ATmega128 can be operated at frequencies up to 16MHz, where one instruction cycle equals one clock cycle.

Given the limited processor resources, we chose to focus our implementation efforts on a small memory footprint using performance optimizations applicable to small devices without significant increases in either code size or memory usage. For ECC, we implemented point multiplication for three SECG-standardized elliptic curves, secp160r1, secp192r1, and secp224r1, including optimized squarings and the techniques described in section 2.2. Inversion was implemented with the algorithm proposed by Chang Shantz [5]. We evaluated the three multiplication strategies with respect to processor capabilities. The CC1010 can access only one bank of 8 registers at a time, where switching register banks requires multiple instruction cycles. Looking at the hybrid multiplication strategy, at least 7 registers are required for the smallest column width of $d = 2$ and two registers are needed to store pointer registers exceeding the number of available registers. We therefore resolved to implementing the row-wise multiplication strategy and unrolled parts of the inner multiplication loop[4]. On the ATmega128, the hybrid multiplication method can be applied with a column width of up to $d = 6$ requiring 19 registers. We chose $d = 5$ for secp160r1 accomodating a 20-byte operand size and $d = 6$ for secp192r1 and secp224r1.

For RSA, we implemented RSA-1024 on both processors and RSA-2048 on the ATmega128 incorporating the optimizations described in section 2.1. The CC1010 implementation of Montgomery multiplication uses row-wise multiplication, where the ATmega128 implementation employs the hybrid strategy using

[4] A later analysis showed that implementing the column-wise strategy would save 9.5% cycles in the inner multiplication loop by reducing the number of memory accesses.

the maximal column width of $d = 6$. Since the operand word size of 64 bytes for RSA-1024 with CRT is not a multiple of $d = 6$, the implementation performs a 528×528-bit Montgomery multiplication, where optimizations could be made at the cost of increased code size. For the RSA public-key operations we used a small exponent of $e = 2^{16} + 1$.

Table 2 summarizes performance, memory usage, and code size of the ECC and RSA implementations. For both the CC1010 and the ATmega128, ECC-160 point multiplication outperforms the RSA-1024 private-key operation by an order of magnitude and is within a factor of 2 of the RSA-1024 public-key operation. Due to the performance characteristics of Montgomery reduction and pseudo-Mersenne prime reduction, this ratio favors ECC-224 even more when compared to RSA-2048.

For point multiplication over secp160r1, over 77% of the execution time on the ATmega128 and over 85% of the execution time on the CC1010 are spent on multiple-precision multiplications and squarings not including reduction. This underlines the need for focusing optimization efforts primarily on the inner multiplication and squaring loops. Confirming this observation, we found that an optimized implementation on the CC1010 that unrolled loops for addition, subtraction, reduction and copy operations required 35% more code space while decreasing execution time by only 3%. Our numbers further show that on processor architectures with small word sizes, the use of pseudo Mersenne primes reduces the time spent on reduction to a negligible amount. Replacing the column-wise with the hybrid method, we measured a performance improvement for ECC point multiplication of 24.8% for secp160r1 and 25.0% for secp224r1 on the ATmega128. Non-adjacent forms accounted for an 11% performance increase on both devices. Comparing the memory requirements, it is interesting to note that while modular exponentiation requires relatively little memory, a full RSA implementation with CRT requires additional routines and several precomputed constants significantly increasing the memory requirements.

Table 3 shows the instruction decomposition for a 160-bit multiplication and a 512/528-bit Montgomery multiplication on both platforms. Looking at the amount of time spent on arithmetic operations, the small register set and the single destination register for arithmetic operations lead to a low multiplication efficiency on the CC1010. Multiplication and addition instructions account for only 38.2% of a 160-bit multiplication and 28.7% for a 512-bit Montgomery multiplication. Despite the high multiplication cost of 5 instruction cycles, register pressure results in frequent memory accesses and a large overhead of non-arithmetic instructions. In comparison, 69.6% of the time for a 160-bit multiplication and 72.6% of the time for a 528-bit Montgomery multiplication is spent on arithmetic instructions on the ATmega128. This increase in efficiency can be mostly attributed to the large register file and variable destination registers for arithmetic operations. Furthermore, the homogeneous memory architecture and post-increment and pre-decrement functionality for memory operations lead to performance advantages for large key sizes.

We expect that performance improvements for ECC and RSA could be achieved by employing window techniques for point multiplication / modular exponentiation and by using Karatsuba Ofman multiplication. However, these techniques would lead to significant increases in data memory usage and code size and add to the complexity of the implementation.

5 Instruction Set Extensions

Table 3 shows that addition and multiplication instructions account for the majority of the execution time for both processors. Significant performance improvements can be achieved by combining one multiplication and two additions into one instruction as proposed by Großschädl [9]. We refer to this instruction as "MULACC" and define it to perform the following operation on a source register r_s, a destination register r_d, a fixed architectural register r_c and a non-architectural register exc (all of bit-width k):

$$\begin{aligned} \mathrm{MULACC}\, r_d, r_s :\quad & r_d && \leftarrow (r_s * r_c + exc + r_d)[k-1..0] \\ & exc && \leftarrow (r_s * r_c + exc + r_d)[2k-1..k] \end{aligned} \tag{3}$$

MULACC multiplies the source register r_s with an implicit register r_c, adds registers exc and r_d and stores the lower k bits of the $2k$-bit result in register r_d. The upper k bits are stored in register exc, from where they can be used in a subsequent MULACC operation. We refer to exc as the "extended carry register" since its function resembles the carry bit used for additions. Applied to the row-wise or hybrid multiplication strategy, MULACC can generate a partial product $a_j * b_i$, add the upper k bits of the previous partial product $a_{j-1} * b_i$, add k bits from an accumulator register and store the result in the accumulator register in a single instruction. Since MULACC uses only two variable registers and r_c, exc are fixed, it is compatible with both the 8051 and AVR instruction sets. Table 3 shows the instruction decomposition for a 160-bit multiplication and 528-bit Montgomery multiplication using the MULACC instruction on the ATmega128. Implemented as a 2-cycle instruction, MULACC reduces the execution time of a 160-bit multiplication by more than 36% resulting in a total reduction of point multiplication time of 27.6% to 0.59s. The execution time for 528-bit Montgomery multiplication is reduced by 39%. MULACC further reduces the number of registers needed for the inner multiplication loop such that the hybrid multiplication method could be implemented with a column width of $d = 8$, which would result in even higher performance gains. On the CC1010, we measured a reduction in execution time of 39% to 2.78s for secp160r1 point multiplication and 38% to 33.06s for 512-bit Montgomery exponentiation.

6 Conclusions

We compared elliptic curve point multiplication over three SECG/NIST curves secp160r1, secp192r1, and secp224r1 with RSA-1024 and RSA-2048 on two 8-bit processor architectures. On both platforms, ECC-160 point multiplication

Chipcon CC1010						
			160x160 mult.		**512x512 Montg. mult.**	
Instruction type	**Opcodes**	**Cycles/instr.**	**Instr. cycles**	**%**	**Instr. cycles**	**%**
Register swap	XCH A, B	2	2280	24.46	32512	12.60
Multiplication	MUL	5	2000	21.46	41280	16.00
Addition	ADD/ADDC	1	1560	16.74	32723	12.68
Data stores	MOV ADDR, RX	1	1220	13.09	17057	6.61
Data loads	MOV RX, ADDR	1	1025	11.00	41938	16.25
Pointer inc./dec.	INC/DEC	1	895	9.60	33656	13.04
Dec. + branch	DJNZ	3	297	3.19	24957	9.67
Data loads (ext.)	MOVX	2	40	0.43	16934	6.56
Data stores (ext.)	MOVX	2	0	0.00	16678	6.46
Other			4	0.04	307	0.12
Total			9321	100.00	258042	100.00
Time @ 14.7456MHz			2.53ms		70.00ms	
ATmega128						
			160x160 mult.		**528x528 Montg. mult.**	
Instruction type	**Opcodes**	**Cycles/instr.**	**Instr. cycles**	**%**	**Instr. cycles**	**%**
Addition	ADD/ADC	1	1360	43.79	29766	45.67
Multiplication	MUL	2	800	25.76	17556	26.94
16-bit Register move	MOVW	1	335	10.79	7262	11.14
Data loads	LD/LDI	2	334	10.75	6169	9.47
Data stores	ST	2	80	2.58	524	0.80
Jumps	RJMP/IJMP	2	66	2.12	0	0.00
Function calls/rets	CALL/RET	4	0	0.00	1452	2.23
Other			131	4.22	2442	3.75
Total			3106	100.00	65171	100.00
Time @ 8MHz			0.39ms		8.15ms	
ATmega128 with MULACC instruction						
			160x160 mult.		**528x528 Montg. mult.**	
Instruction type	**Opcodes**	**Cycles/instr.**	**Instr. cycles**	**%**	**Instr. cycles**	**%**
Multiply-accumulate	MULACC	2	960	48.34	20328	51.27
Data loads	LD/LDI	2	334	16.82	6169	15.56
Addition	ADD/ADC	1	320	16.11	6292	15.87
Data stores	ST	2	80	4.03	524	1.32
Jumps	RJMP/IJMP	2	66	3.32	0	0.00
Function calls/rets	CALL/RET	4	0	0.00	1452	3.66
Multiplication	MUL	2	0	0.00	924	2.33
16-bit Register move	MOVW	1	15	0.76	2	0.01
Other			211	10.62	3960	9.99
Total			1986	100.00	39651	100.00
Time @ 8MHz			0.25ms		4.96ms	
Time reduction			36.06%		39.16%	

Table 3. *Decomposition of 160x160-bit multiplication and 512x512/528x528-bit Montgomery multiplication on the Chipcon CC1010 and the ATmega128.*

Reduction for 160-bit multiplication and the conditional subtraction of the prime for Montgomery multiplication are not included in the instruction counts.

outperforms the RSA-1024 private-key operation by an order of magnitude and is within a factor of 2 of the RSA-1024 public-key operation.

We presented a novel multiplication algorithm that significantly reduces the number of memory accesses. This algorithm led to a 25% performance increase for ECC point multiplication on the Atmel AVR platform.

Our measurements and analysis led to fundamental observations: The relative performance of ECC over RSA increases as the word size of the processor decreases. This stems from the fact that the complexity of addition, subtraction and optimized reduction based on sparse pseudo-Mersenne primes grows linearly with the decrease of the word size whereas Montgomery reduction grows quadratically. As a result, ECC point multiplication on small devices becomes comparable in performance to RSA public-key operations and we expect it to be higher for large key sizes.

In contrast to Hasegawa et al. and Woodbury, Bailey and Paar, our observations do not support the claim that field primes chosen specifically for a particular processor architecture or OEFs lead to significant performance improvements over prime fields using pseudo-Mersenne primes as recommended by NIST and SECG. Using pseudo-Mersenne primes as specified for NIST/SECG curves, more than 85% of the time for secp160r1 point multiplication on the 8051 architecture and more than 77% on the AVR architecture was spent on integer multiplication not including reduction. Therefore, further optimizing reduction would not lead to significant performance improvements. Woodbury, Bailey and Paar represent field elements $GF((2^8 - 17)^{17}$ as polynomials with 17 8-bit integer coefficients. Polynomial multiplication in this field requires the same number of 8x8-bit multiplications as 17-byte integer multiplication. The algorithm for polynomial multiplication corresponds to integer multiplication using the column-wise method, where optimized reduction is performed at the end of each column. The hybrid or row-wise methods cannot be applied such that we expect the performance of ECC over OEFs to be lower on architectures such as the Atmel AVR.

We plan to continue our work on small devices towards a complete lightweight implementation of the security protocol SSL/TLS.

References

1. Atmel Corporation. http://www.atmel.com/.
2. D. V. Bailey and C. Paar. Optimal Extension Fields for Fast Arithmetic in Public-Key Algorithms. In *Advances in Cryptography — CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 472–485. Springer-Verlag, 1998.
3. Ç. K. Koç. High-Speed RSA Implementation. Technical report, RSA Laboratories TR201, November 1994.
4. Certicom Research. SEC 2: Recommended Elliptic Curve Domain Parameters. Standards for Efficient Cryptography Version 1.0, September 2000.
5. S. Chang Shantz. From Euclid's GCD to Montgomery Multiplication to the Great Divide. Technical report, Sun Microsystems Laboratories TR-2001-95, June 2001.
6. Chipcon AS. http://www.chipcon.com/.

7. H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. In *ASIACRYPT: Advances in Cryptology*, volume 1514 of *Lecture Notes in Computer Science*, pages 51–65. Springer-Verlag, 1998.
8. Crossbow Technology, Inc. http://www.xbow.com/.
9. J. Großschädl. Instruction Set Extension for Long Integer Modulo Arithmetic on RISC-Based Smart Cards. In *14th Symposium on Computer Architecture and High Performance Computing*, pages 13–19. IEEE Computer Society, October 2002.
10. D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer-Verlag, 2004.
11. T. Hasegawa, J. Nakajima, and M. Matsui. A practical implementation of elliptic curve cryptosystems over GF (p) on a 16-bit microcomputer. In *Public Key Cryptography PKC '98*, volume 1431 of *Lecture Notes in Computer Science*, pages 182–194. Springer-Verlag, 1998.
12. Y. Hitchcock, E. Dawson, A. Clark, and P. Montague. Implementing an efficient elliptic curve cryptosystem over GF(p) on a smart card. *ANZIAM Journal*, 44(E):C354–C377, 2003.
13. A. Karatsuba and Y. Ofman. Multiplication of Many-Digital Numbers by Automatic Computers. *Doklady Akad. Nauk*, (145):293–294, 1963. Translation in Physics-Doklady 7, 595-596.
14. F. Morain and J. Olivos. Speeding up the computations on an elliptic curve using addition-subtraction chains. *Theoretical Informatics and Applications*, 24:531–543, 1990.
15. National Institute of Standards and Technology. Recommended Elliptic Curves for Federal Government Use, August 1999.
16. H. Pietiläinen. *Elliptic curve cryptography on smart cards.* Helsinki University of Technology, Faculty of Information Technology, October 2000. Master's Thesis.
17. A. D. Woodbury, D. V. Bailey, and C. Paar. Elliptic Curve Cryptography on Smart Cards without Coprocessors. In *The Fourth Smart Card Research and Advanced Applications (CARDIS2000) Conference*, September 2000. Bristol, UK.

A Algorithm for Hybrid Multiplication

The two outer nested loops describe column-wise multiplication and the two inner nested loops describe row-wise multiplication. Multiplicand and multiplier are located in memory locations `mem_a` and `mem_b` and are temporarily loaded into registers $a_{d-1}, \ldots, a_0$ and b. The result is accumulated in registers $r_{2d-1+\lceil log2(n/d)/k \rceil}, \ldots, r_0$, where the lower d registers are stored to result memory location `mem_c` at the end of each column.

```
Input:
n                       : operand size in words
d                       : column width
mem_a [⌈ n/d ⌉*d-1..0] : multiplicand A
mem_b [⌈ n/d ⌉*d-1..0] : multiplier B

Output:
mem_c [⌈ n/d ⌉*2d-1..0] : result C = A * B

for i=0 to ⌈ n/d ⌉-1
  for j=0 to i
    (a_{d-1},..., a_0) = mem_a[(i-j+1)*d-1..(i-j)*d]
    for s=0 to d-1
      b = mem_b[j*d+s]
      for t=0 to d-1
        (r_{2d-1+⌈ log2(n/d)/k ⌉},..., r_0) = (r_{2d-1+⌈ log2(n/d)/k ⌉},..., r_0) +
                                     a_t * b * 2^{k*(t+s)}
  mem_c[(i+1)*d..i*d] = (r_{d-1},..., r_0)
  (r_{d-1+⌈ log2(n/d)/k ⌉},..., r_0) = (r_{2d-1+⌈ log2(n/d)/k ⌉},..., r_d)
  (r_{2d-1+⌈ log2(n/d)/k ⌉},..., r_d) = 0
for i=⌈ n/d ⌉ to 2⌈ n/d ⌉-2
  for j=i-⌈ n/d ⌉+1 to ⌈ n/d ⌉-1
    (a_{d-1},..., a_0) = mem_a[(i-j+1)*d-1..(i-j)*d]
    for s=0 to d-1
      b = mem_b[j*d+s]
      for t=0 to d-1
        (r_{2d-1+⌈ log2(n/d)/k ⌉},..., r_0) = (r_{2d-1+⌈ log2(n/d)/k ⌉},..., r_0) +
                                     a_t * b * 2^{k*(t+s)}
  mem_c[(i+1)*d ..i*d] = (r_{d-1},..., r_0)
  (r_{d-1+⌈ log2(n/d)/k ⌉},..., r_0) = (r_{2d-1+⌈ log2(n/d)/k ⌉},..., r_d)
  (r_{2d-1+⌈ log2(n/d)/k ⌉},..., r_d) = 0
 mem_c[(i+1)*d ..i*d] = (r_{d-1},..., r_0)
```

B Algorithm for Montgomery Reduction

The algorithm below describes Montgomery reduction of a $2m$-bit multiplication result $C' = A * B$ of two m-bit numbers A and B to $C = C' * r^{-1} \mod n$ on a processor with a k-bit word size.

```
n' = -1/n mod 2^k
for i=0 to ⌈ m/k ⌉-1
   s=C'*n' mod 2^k
   C'=C'+ s * n    // last k bits of C' become 0
   C'=C' >> k      // division by 2^k
if C'>=n
   C'=C'- n
return C=C'
```