Efficient Implementation of Public Key Cryptosystems on Mote Sensors (Short Paper)

Haodong Wang and Qun Li

Department of Computer Science
College of William and Mary
{wanghd, liqun}@cs.wm.edu

Abstract. We report our implementation of the RSA and ECC public-key cryptosystem on Berkeley Motes. We detail the implementation of 1024-bit RSA and 160-bit ECC cryptosystems on MICA mote sensors. We have achieved the performance of 0.79s for RSA public key operation and 21.5s for private operation, and 1.3s for ECC signature generation and 2.8s for verification. For comparison, we also show our new ECC implementation on TelosB motes with a signature time 1.60s and a verification time 3.30s. For the detailed description of the implementation, we refer to our technical report[13].

1 Introduction

Public-key cryptography has been used extensively in data encryption, digital signature, user authentication, access control[12,14], etc. Compared with the symmetric key based schemes proposed for sensor networks, public-key cryptography is more flexible requiring no complicated key pre-distribution and no pairwise key sharing negotiation. It is a popular belief, however, in sensor network research community that public-key cryptography, such as RSA and Elliptic Curve Cryptography (ECC), is not practical because the required computational intensity is prohibitive for sensors with limited computation capability and extremely constrained memory space. The nascent exploration has already disabused of this misconception. The recent progress in ECC and RSA implementation on Atmel ATmega128[3], a CPU of 8Hz and 8 bits, shows that public-key cryptography is feasible for sensor network security related applications. This paper describes our implementation of 1024-bit RSA cryptosystem and 160-bit ECC cryptosystem on Motes of MICA2 family with a comparison of our new ECC implementation on TelosB motes.

The major operations in RSA and ECC cryptosystems are large integer arithmetics over the finite field. To efficiently perform RSA and ECC exponentiations on the low-power CPU of sensor motes, it is essential to optimize the expensive large integer operations, especially multiplication and reduction. Since most CPU cycles are consumed in these two integer operations, the efficiency of these two integer operation modules directly determines the performance of the encryption and decryption. Low-power sensor microcontroller usually has a very limited number of registers (32 8-bit registers in ATmega 128). Thus the time

P. Ning, S. Qing, and N. Li (Eds.): ICICS 2006, LNCS 4307, pp. 519–528, 2006.

for long integers to be loaded from or stored to memory is not negligible and the memory accesses have to be optimized for better performance. In this paper, we adopt the hybrid multiplication method [4], which is a very effective way to reduce the number of memory accesses. To precisely control the register and memory operations, we implement this module in assembly language. Our experiments demonstrate that the hybrid multiplication is at least 7 times faster than the conventional multi-precision multiplication programmed in C language. The modular reduction can also be optimized under certain conditions. For example, when the modulus is a pseudo-Mersenne number, the reduction can be greatly optimized and be finished more than 10 times faster than the classic long division method.

In addition to the optimization of the big integer operations, RSA and ECC can be further optimized. In RSA, Montgomery reduction can be applied to efficiently calculate the RSA exponentiation, and Chinese Remainder Theorem (CRT) can be used to reduce the exponent sizes and speed up the RSA exponentiation for up to 4 times. In ECC, we apply a mixed coordinate, the combination of Affine coordinate and Jacobian coordinate, to accelerate ECC exponentiation by avoiding operations such as inversions or reducing the amount of operations such as multiplication and squaring.

Our experiments show that both RSA and ECC can efficiently run on MICAz motes. For RSA, it takes 0.79 second to do a public key operation, and 21.5 seconds to perform a private key operation. For ECC, it takes 1.3 seconds to generate a signature, and 2.8 seconds to perform a signature verification. For our new ECC implementation on TelosB, the signature time and verification time are 1.60s and 3.30s respectively. It is possible to further reduce the computation time by using extended instruction set adopted in [4]. Our experiment results demonstrate that most operations in RSA and ECC are feasible for sensor network security applications.

2 Implementation

We have implemented RSA and ECC cryptosystems on MICAz motes, powered by ATmega128 microcontroller. The ATmega128 incorporates an 8MHz, 8-bit RISC CPU, 128K bytes programmable flash memory (ROM) and 4K bytes SRAM. This architecture provides 133 powerful instructions and 32×8 general purpose registers. Besides, ATmega128 also features an on-chip multiplier. In this section, we first describe the optimized large integer operation modules, which can be used for both RSA and ECC cryptosystems. Then we focus on the protocol related optimizations specifically for RSA and ECC, respectively. For ECC implementation, without further clarification, we concentrate on SECG recommended 160-bit elliptic curve: secp160r1.

2.1 Large Integer Operations

We have implemented a suite of large integer arithmetic operations, including addition, subtraction, shift, multiplication, division and modular reduction.

Among three different multiplication implementations [4,8,7], we have chosen to use Hybrid Multiplication proposed in [4]. We have implemented Hybrid multiplication in assembly language with column width $d = 4$, which requires 9 accumulator registers, 5 operand registers, 6 pointer registers, and others for temporary storage and loop control. For the comparison purpose, we also implement a standard multi-precision multiplication program in C language. Our experiments show the standard C program needs $122.2ms$ to finish the multiplication between two 128-byte integers, while it only takes $17.6ms$ for our Hybrid multiplication to do the same computation, which is more than 7 times faster.

Squaring is a special case of the multiplication, which has the same the multiplicand and the multiplier. Given an m-bit large integer $A = (A_1, A_0)$, where A_1, A_0 are two halves, $A^2 = A_1A_1 \times 2^m + 2A_1A_0 \times 2^{m/2} + A_0A_0$. Therefore, we can take advantage of the fact that A_1A_0 only needs to be calculated once. Compared with the multiplication, the optimized squaring can reduce the computational complexity up to 25%.

For Modular Reduction, We choose the classic long division method to implement this operation. Fortunately, the number of this type of modular reduction is very limited, it does not affect the overall performance much. The long division producer reduces the remainder by one byte in each iteration. In ECC cryptosystem, we choose to use pseudo-Mersenne primes as specified in NIST/SECG curves, the modular reduction can be optimized by conducting a fixed number of integer additions.

Modular inversion is used in both ECC and RSA. For ECC operation, we adopt an efficient Great Divide scheme [11]. For RSA operation, we use the classic Extended Euclidean Algorithm.

2.2 RSA Operations

In our first RSA implementation, it takes 4.6 seconds to finish the public key operation and 389 seconds to do a private key operation. To reduce the computational time, we have implemented the following two optimizations.

Montgomery Reduction. Montgomery reduction [9] is a method to efficiently perform the modular reduction without doing expensive division. For example, suppose we want to compute T modulo N, the algorithm says it is easy to compute $TR^{-1} \pmod N$ (without any division), where R is a radix ($R > N$) and co-prime to N. We do not validate this algorithm in this paper. Interested reader may refer to [9] for details. Having implemented the Montgomery reduction module, the performance of RSA public key and private key operations have been improved significantly to 1.2s and 82.2s, respectively.

Chinese Remainder Theorem (CRT). The complexity of the exponentiation in RSA largely depends on the the size of modulus n and the exponent (either public key or private key). Chinese Remainder Theorem (CRT) can be used to effectively reduce the computational complexity of exponentiation by reducing the size of both n and the exponent. With CRT implemented, the

public key operation has been reduce to 0.79s. Correspondingly, the private key operation is reduced to 21.5s, approximately 1/4 of the time before doing CRT.

2.3 ECC Operations

Here we briefly discuss our optimizations for ECC operations.

ECC Addition and Doubling. The fundamental ECC operation is point addition and point doubling. The point multiplication can be decomposed to a series of addition and doubling operations. As discussed in previous section, point addition and doubling in Affine coordinate require integer inversion, which is considered much slower than integer multiplication. Cohen *et al.* showed that these operations in Projective coordinate and Jacobian coordinate yield better performance [1]. They further found addition and doubling in mixed coordinate, with the combination of Modified Jacobian coordinate and Affine coordinate, lead to the best performance [2]. As the result, point doubling operation reduces to 4 multiplications and 4 squaring, and the computational complexity of the point addition reduces to 8 multiplications and 3 squaring. Our experiments show that the performance of point multiplication improves around 6% compared with our previous implementation in Jacobian coordinate.

Modular Reduction on ECC Curve. Recall that modular reduction has to be applied after every large integer multiplication, it is also a performance critical operation. By taking advantage of pseudo-Mersenne primes specified in SECG curves, the complexity of the modular reduction operation can be reduced to a negligible amount.

Further Optimization. Examining the computational complexity, we notice that point addition is more expensive than point doubling. We adopt Non-adjacent forms (NAFs) [10] and sliding window method [5] in our implementation. According to our experiments, point multiplication with NAFs contributes at least 5% performance improvement. For sliding window, we select window size $s = 4$. Correspondingly, there are 16 entries in the partial result table. Our experiments show sliding window method is more effective than NAFs for fixed point multiplication, the performance of sliding window method is more than 10% better than that of NAFs.

3 Experiments and Performance Evaluation

We have implemented the 1024-bit RSA and the 160-bit ECC security primitive on MICAz motes, the latest sensor motes of the MICA family from Crossbow. Our experiments show that the public key operation (17-bit public key) only takes 0.79s and private key operation takes 21.5s. For the ECC operations, it takes 1.3 seconds to generate a signature and 2.8 second to do a signature verification. Considering that RSA verification normally happens at sensor side, and expensive signature generation is done by powerful devices, such as PDAs, we conclude both RSA and ECC are practical for small sensor nodes.

3.1 RSA Evaluation

In this subsection, we describe the experimental performance of 1024-bit RSA on our MICAz motes. We first present our experimental results and related issues during the implementation. We then give the performance analysis to quantify the computational complexity.

Experimental Results and Implementation Challenge. In the experiment, we randomly select two 512-bit prime number as p and q. For the public key operation, we choose a small exponent of $e = 2^{16} + 1$, which is commonly used value for e. Our program uses 15,832 byte code size and 3,224 byte data size. Compared with RSA implementation in [4], our code size is much larger because of the assignments of precomputation values during initialization stage. Our implementation spends 0.79s to finish a publick key operation and 21.5s to do a private key operation.

The biggest challenge in implementing 1024-bit RSA on MICAz motes is the memory constraint. MICAz mote only has 4KB RAM, which is the total space for data and program stack. Since the operands in 1024-bit RSA are mostly 128 integers, the subroutines, such as modular reduction, Extended Euclidean Algorithm and Montgomery reduction, have to reserve considerable amount of memory space for storing temporary results. In addition, for optimization purpose, a number of pre-computations are required. In our program, 1152 bytes of memory are used for storing system parameters, such as p, q and n, and precomputation results, such as R_p, R_q in CRT. Therefore, attentions need to be paid not to waste any memory usage. In practice, we have adopted two methods to save the memory space. First, we declare more global variables. The idea is to share the memory space among different subroutines in each module. Note this method is only good for those subroutines do not call each other. Otherwise the intermediate data will be lost. Second, we conduct every possible precomputation so that some modules may not be required during the RSA operation in the real time. For example, the Extended Euclidean algorithm is only used to find the public/private key pairs and to precompute the parameters used in Montgomery reduction. Removing this module saves us 1K data space.

Performance Analysis. To analyze the computational complexity distribution among the components in RSA exponentiation, we profile the execution time of multiplication, squaring, and modular reduction modules, the three most time consuming operations in RSA exponentiation. The profiling information is shown in Table 1.

Our analysis assumes that all optimization schemes have been applied in RSA exponentiation. To simplify the presentation, we denote “MUL” as, large integer multiplication, and let “SQR” be large integer squaring, and let “MOD” be large integer modular reduction. An ”m/n” MOD means a MOD operation for a m-byte integer over a modulus with n-bytes. For example, 128/64 MOD denotes a modular reduction of a 128 byte integer with a 64 byte modulus.

Table 1. Execution time profiles of some important modules

Module	Operand Sizes (bytes)	Execution Time (ms)
MUL.	128 by 128	17.1
MUL.	64 by 64	4.48
SQR.	128 by 128	14.1
SQR.	64 by 64	3.87
MOD.	256/128	132
MOD.	192/128	74
MOD.	128/64	40

Let us consider an example of RSA operation to calculate $M = C^x \pmod n$, where x can be either public key or private key. Following the CRT algorithm, we first do two MODs to calculate C_p and C_q. Then, we conduct two Montgomery reductions to get M_p and M_q. Finally, two MULs, one MODs and one addition are required to compute M. Note the last two steps in CRT, which requires 2 MODs, can be simplified by doing addition first and then only one MOD. Except the Montgomery reduction, both public key and private key operation need to do two 128/64 MODs, two 128×128 MULs, one 192/128 MODs operations, which totally account for $2 \times 40 + 2 \times 17.1 + 74 = 188.2ms$.

The difference of execution time between public key and private key operations is at exponentiation part. Each Montgomery reduction requires two 64×64 MULs, one 128-byte addition and possible another 128-byte subtraction. The cost of addition and subtraction can be ignored. Therefore, the execution time of each Montgomery reduction is $2 \times 4.48 = 8.96ms$. Since we choose the public key to be $2^{16} + 1$, there are totally 16 64×64 SQRs and 1 64×64 MUL in the exponentiation. According to Table 1, the total time for SQRs and MUL with Montgomery reduction should be $16 \times 3.87 + 4.48 + 17 \times 8.96 = 218.7ms$. In addition, two 128/64 MODs are needed to convert operands between integer and N-residue before and after each exponentiation. For CRT optimization, we need to do two 512-bit exponentiations. Therefore, the exponentiation execution time for public key operation is $2 \times (218.7 + 2 \times 40) = 597.4ms$. Combined with the rest operations in CRT, the public key operation consumes $594.4 + 188.2 = 782.6ms$, which matches our test result very well.

For the private key operation, the number of SQRs is 511 (after CRT) in each reduced exponentiation. The number of MULs depends on the Hamming weight of the exponent. Our experiment shows the average Hamming weight of D_p and D_q of our private key is 278. Hence, there are 277 MULs required in each exponentiation. Therefore, the execution time for each exponentiation is $511 \times 3.87 + 277 \times 4.48 + 788 \times 8.96 = 10279ms$. Since the exponentiation execution time in private key operation overwhelmingly dominates other operations, we only need to consider the execution time of exponentiations only. Two such exponentiations consumes 20.5 seconds, closely matching our experiment result of 21.5s.

3.2 ECC Evaluation

In this subsection, we first present the performance of our implementation. Then we give an overall analysis to quantify the computation complexity.

The Performance of ECC Implementation. In experiments, we measure execution time and code size of our implementation. We choose secp160r1 as the elliptic curve in all experiments. We use the embedded system timer (921.6kHz) to measure the execution time of major operations in ECC, such as point multiplication, point addition and point doubling.

We first test point multiplication operation, which is comprised of point addition and doubling. We consider two cases in point multiplication. One is multiplying large integer with a fixed point(base point), and the other one is with a random point. Fixed point multiplication allows for optimization by precomputing. We apply sliding window technique[6] and set window size to 4, i.e., precomputing $2^4 - 1 = 15$ points. In experiments, we randomly generate 20 large integers to multiply with the point and take the average execution time as the result.

Since ECC point multiplication consists of addition and doubling operations, we further evaluate these two operations separately. We generate random points and large integers for tests. Since a single operation takes very little time, to reduce the error of clock inaccuracy, we measure 100 operations every round and take the average value.

Table 3 shows the experimental results of execution time. Point addition and doubling of our implementation is superior to the other two implementations, which results in a faster point multiplication.

Next, we implement ECDSA signature scheme. The experimental results are shown in Table 3. In fact, signing a message is mainly a fixed point multiplication. As we can see, the signature time is very close to the time consumed in fixed point multiplication. On the other hand, verification of ECDSA consists of one fixed point multiplication and one random point multiplication. Therefore, the performance of the verification is roughly the summation of one fixed point multiplication and one random point multiplication.

Table 2 presents the code size of the ECC implementation. The ECC library itself only uses 18.8KB ROM and 1.36KB RAM. However, ECDSA consumes 56.4KB ROM and 1.7KB RAM. The reason is that we add SHA1 hash function and radio communication module in the ECDSA package, where SHA-1, occupying more than 30KB memory space, takes a large portion of the code size.

Table 2. ECC implementation code sizes

	ECC library		ECDSA	
	ROM	RAM	ROM	RAM
ECC	18.8k	1.36k	56.4k	1.7k

A Performance Anatomy of ECC Point Multiplication on MICAz. Since ECC point multiplication dominates the computational complexity in ECC signature and verification, we are curious to learn the performance anatomy in ECC point multiplication.

This analysis is based on 160-bit ECC curves. We use secp160r1 as the example. We also assume 4-bit sliding window method is used, and partial results are precomputed. The computational cost for each window unit is 4 point doubling and 1 point addition. Given a 161 bit private key, there are 41 window units. Totally , 164 point doubling and 41 point additions are required to finish 1 point multiplication.

Large (160-bit) integer multiplication, squaring and reduction are the most expensive operations in point doubling and point addition. To learn the amount of time contributed by the above three operations in a fix point multiplication. We first individually test the performance of large integer multiplication, squaring and reduction. Our results show that it takes $0.47ms, 0.44ms$ and $0.07ms$ to perform a 160×160 multiplication, squaring and reduction, respectively. Next, we count the the number of each operation required in a point multiplication. Since we adopt the mixed coordination (the combination of Jacobian coordinate and Affine coordinate), each point addition requires 8 large integer multiplications and 3 large integer squaring, and each point doubling requires 4 large integer multiplications and 4 large integer squaring. In addition, each multiplication, squaring or shifting operation has to be followed by a modular reduction. Our program shows the point addition requires 12 modular reductions, and the point doubling requires 11 modular reductions. In total, each point multiplication costs $164 \times 4 + 41 \times 8 = 984$ large integer multiplications, $164 \times 4 + 41 \times 3 = 779$ large integer squaring and $164 \times 11 + 41 \times 12 = 2,296$ large integer modular reductions. Plugging in the results of the individual tests, we get the total amount of time consumed on the three operations is 0.97s, roughly 78.2% of the total time to do a fix point multiplication. The rest of 21.8% of the time is spent on various operations, including inversion operation (to convert the Jacobian coordinate to Affine), addition, subtraction, shifting and memory copy, etc. Based on above analysis, we believe the performance of ECC operations on MICAz can be further improved by more refined and careful programming.

Performance Comparison. In the last part of the evaluation, we first investigate the performance difference of our cryptosystem implementation on different sensor platforms. Then we compare the performance of our implementation with existing research result [4] and give the possible explanation of the performance gap.

To learn the performance of the public key cryptosystems on different sensor platforms, we have revamped our previous ECC implementation on TelosB mote[14]. We summarize the performance comparison in Table 3. It clearly shows that the performance of ECC operation on MICAz is slightly better than that on TelosB, even though TelosB is equipped with a 8MHz, 16-bit CPU. After a careful and tedious investigation, we found the performance degradation on TelosB is due to the following two reasons. First, the 8MHz CPU (MSP430) frequency

Table 3. The comparison of ECC execution Time on both mote platform operations, including fixed point multiplication (FPM), random point multiplication (RPM), point addition (PAdd) and point doubling (PDbl) and ECDSA signature generation (SIGN), verification (VERIFY) time

	FPM	RPM	PAdd	PDbl	SIGN	VERIFY
MICAz	1.24s	1.35s	6.2ms	5.8ms	1.35s	2.85s
TelosB	1.44s	1.60s	7.3ms	7.0ms	1.60s	3.32s

on TelosB is just a nominal value. In reality, the maximum CPU clock rate is actually 4MHz. Second, the hardware multiplier in MSP430 CPU uses a group of special peripheral registers which are located outside of MSP430 CPU. As the result, it takes MSP430 eight CPU cycles to perform an unsigned multiplication, while it at most takes four cycles to do the same operation in Atmega CPU. The above two reasons explain why TelosB cannot perform better than MICAz.

We also compare our ECC performance with the result in [4]. Gura *et al.* implemented the ECC (the same curve) on Atmega128 CPU, which is the same CPU used on MICAz mote. Their result, 0.81s for a random point multiplication, is about 40% faster than 1.35s of our result. We notice that the time for their 160×160 multiplication is 0.39ms, roughly 17% faster than our 0.47ms. In general, we believe their code is more polished and optimized in many aspects than our code. Furthermore, Our code is implemented in TinyOS, and mostly written with NesC (except several critical large integer operations), which could introduce additional CPU cycles.

4 Conclusion

In this paper, we present a number of optimization schemes to efficiently implement the public key cryptosystems in small, less-powerful sensor devices. We implement 1024-bit RSA and 160-bit ECC on Mica motes. Our experiments demonstrate that the public key cryptography is promising for sensors. Our experiments show that the times for ECC signature generation and verification are 1.3s and 2.8s respective for Mica motes, and 1.6s and 3.3s for TelosB motes. For RSA implementation, we have achieved 0.79s for public key operation and 21.5s for private operation on Mica motes. We believe the performance can be improved by more careful programming or using more powerful sensors.

References

1. H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation. In *ICICS'97*, pages 282–290, Springer-Verlag, 1997.
2. H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. In *ASIACRYPT*, 1998.
3. V. Gupta, M. Millard, S. Fung, Y. Zhu, N. Gura, H. Eberle, and S. Shantz. Sizzle: A standards-based end-to-end security architecture for the embedded internet. In *PerCom*, Kauai, Mar. 2005.

4. N. Gura, A .Patel, A. Wander, H. Eberle, and S. C. Shantz. Comparing elliptic curve cryptography and rsa on 8-bit cpus. In *CHES*, Boston, Aug. 2004.
5. C. K. Koc. High-speed rsa implementation, rsa laboratories technical report tr-201, version 2.0. Nov 22 1994.
6. C. K. Koc. High-speed rsa implementation. In *RSA Lab TR201*, Nov. 1994.
7. A. Liu and P. Ning. Tinyecc: Elliptic curve cryptography for sensor networks. Sept 15 2005.
8. D.J. Malan, M. Welsh, and M.D. Smith. A public-key infrastructure for key distribution in tinyos based on elliptic curve cryptography. In *SECON*, Santa Clara, CA, October 2004.
9. P. Montgomery. Modular multiplication without trial division. *Mathematics of Communication*, 44(170):519–521, April 1985.
10. 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.
11. S. Chang Shantz. From euclid's gcd to montgomery multiplication to the great divide. In *Technical report, Sun Lab TR-2001-95*, June 2001.
12. Haodong Wang and Qun Li. Distributed user access control in sensor networks. In *IEEE DCOSS*, pages 305–320, San Francisco, CA, June 2006.
13. Haodong Wang and Qun Li. Efficient Implementation of Public Key Cryptosystems on MicaZ and TelosB Motes. Technical Report WM-CS-2006, College of William and Mary, October 2006.
14. Haodong Wang, Bo Sheng, and Qun Li. Elliptic curve cryptography based access control in sensor networks. *Int. Journal of Security and Networks*, 1(2), 2006.