WM-CS-2006-07

Efficient Implementation of Public Key Cryptosystems on MICAz and TelosB Motes

Haodong Wang and Qun Li

Oct. 30, 2006

Technical Report WM-CS-2006-07

Efficient Implementation of Public Key Cryptosystems on MICAz and TelosB Motes

Haodong Wang and Qun Li
Department of Computer Science
College of William and Mary
Williamsburg, VA, 23187
E-mail:{wanghd, liqun}@cs.wm.edu

Abstract

Even though symmetric-key scheme, which has been investigated extensively for sensor networks, can fulfill many security requirements, public-key cryptography is more flexible and simple rendering a clean interface for the security component. Against the popular belief that public key scheme is not practical for sensor networks, this technical report describes the RSA and ECC public-key cryptosystem implementation in the real world sensor devices. We detail the implementation of 1024-bit RSA and 160-bit ECC cryptosystems on MICAz sensor motes, the latest product of Crossbow in the MICA family. We evaluate the performance of our implementation by running the public key and the private key operations in RSA cryptosystem, and signature generation and verification in ECC cryptosystem. We have achieved the performance of 0.79s for RSA public key operation and 1.35s for ECC signature generation. For comparison, we show our new ECC implementation on TelosB motes with a signature time 1.55s and a verification time 2.25s. We also explain the reasons that TelosB mote can not perform better than MICAz even though it is equipped with a 16-bit CPU. We believe that the experiment results are encouraging, and RSA and ECC are getting closer to be practially implemented in the sensor motes in the real world.

1 Introduction

Public-key cryptography has been used extensively in data encryption, digital signature, user authentication, etc. Compared with the popular symmetric key based schemes proposed for sensor networks, public-key cryptography provides a more flexible and simple interface requiring no complicated key pre-distribution, no pair-wise 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 not suitable for sensors with limited computation capability and extremely constrained memory space. The nascent exploration has already disabused of this misconception. The recent progress in 1024-bit RSA implementation on Atmel ATmega128, a CPU of 8Hz and 8 bits[4], shows that a public key operation takes less than one second, which proves public-key cryptography is feasible for sensor network security related applications.

This technical report describes our implementation of 1024-bit RSA cryptosystem and 160-bit ECC cryptosystem on MICAz, a latest sensor platform of MICA family from Crossbow. It is of the size of two AA batteries integrating USB programming capability, an IEEE 802.15.4 radio with integrated antenna, a low-power 8-bit MCU. Its detail features include: IEEE 802.15.4/ZigBee compliant RF transceiver, 2.4 to 2.4835 GHz (a globally compatible ISM band), 250 kbps data rate, 8 bit, 8MHz Amtel ATmega microcontroller with

4KB RAM, low current consumption, 128KB programmable ROM, and optional external memory for data collection.

The fundamental 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. In particular, multiplication and reduction are most dominant operations in both RSA and ECC. 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. The low-power sensor microcontroller has very limited number of registers (only 32 8-bit registers in ATmega 128). The large integer operands cannot be loaded into the registers at one time, so that the latency of memory accesses have to be paid for operand loading and storing between registers and memory. The implementation challenge is to reduce the number of such memory accesses. In this technical report, we adopt the hybrid multiplication method [5], 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 optimizations of the big integer operation. RSA and ECC can be further optimized. Montgomery reduction can be applied to efficiently calculate the RSA exponentiation. 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 do ECC exponentiation, so that some expensive operations can be avoided (e.g., inversion) or reduced (e.g., multiplication and squaring).

Our experiments show that both RSA and ECC can efficiently run on MICAz motes. For RSA, it takes 0.79s to do a public key operation, and 21.5s to perform a private key operation. For ECC, it takes 1.35s to generate a signature, and 1.96s to perform a signature verification. It is possible to further reduce the computation time by using extended instruction set proposed in [5]. Our experiment results demonstrate that most operations in RSA and ECC are feasible for sensor network security applications. It also can be inferred by the results that the combination of RSA and ECC has the potential to yield a better performance than that the either single cryptosystem can achieve.

The rest of the technical report is organized as follows. Section 2 briefly introduces RSA and ECC public key schemes. Section 3 gives detail description of several most important optimizations in large integer operation, as well as some specific optimizations designed for RSA and ECC implementations exclusively. Section 4 evaluates the performance of our implementations. Section 5 concludes the technical report.

2 RSA and ECC Background

In this section, we give brief introduction of RSA and ECC public key cryptography. RSA is the most popular security scheme and widely used in security applications. Recently, ECC receives more attentions due to its nice security capability with the much smaller signature size.

2.1 RSA Introduction

In RSA cryptography, a user, say Alice, has two keys: a public key (e) and a private key (d). Alice publishes her public key and keeps private key in secret. When Bob wants to send a message m to Alice, and does not want any other to know the message contents, he just encrypts m by using Alice's public key. Without the private key, it is computationally infeasible for others to decrypt the ciphertext. After receiving the encypted message from Bob, Alice uses her private key to decrypt the message.

The security of RSA scheme is based on the difficulty to factor a large integer (n). Here we briefly go over the key generation procedure and encryption/decryption in RSA. Alice needs to take following steps to get her public key e and private key d.

- Pick two random large prime number p and q, so that $p \neq q$;
- Compute $n = p \times q$;
- Compute the totient: $\phi(n) = (p-1)(q-1)$;
- Choose an integer e as the public key so that $1 < e < \phi(n)$, and e is co-prime to $\phi(n)$;
- Compute the private key $d = e^{-1}$ (mod $\phi(n)$).

To encrypt a message m, Bob computes $c = m^e$ and sends cipher text c to Alice. The decryption for Alice is to raise the value of her private key to the power of the ciphertext c, so that $c^d = (m^e)^d = m^{ed} = m$ (mod n). The decryption procedure works due to following reasons. Because $e \times d \equiv 1$ (mod $(p-1)(q-1)$), we have $e \times d \equiv 1$ (mod $(p-1)$) and $e \times d \equiv 1$ (mod $(q-1)$). Applying *Fermat's little theorem*, we get $m^{ed} \equiv m$ (mod p) and $m^{ed} \equiv m$ (mod q). Applying *Chinese Remainder Theorem* (CRT), we have $m^{ed} \equiv m$ (mod n).

In practice, RSA must be combined with certain padding scheme to defend against security attacks, such as Adaptive Chosen Cipher Text attack. The popular padding schemes include Optimal Asymmetric Encryption Padding (OAEP) and Probabilistic Signature Scheme for RSA (RSA-PSS). For the simplicity, we do not cover the padding scheme implementation in this technical report.

2.2 ECC Introduction

In this section, we briefly give a background introduction about elliptic curve cryptography, and corresponding elliptic curve Digital Signature Algorithm.

2.2.1 Elliptic Curve Cryptography

In recent years, ECC has attracted much attention as the security solutions for wireless networks due to the small key size and low computational overhead. For example, 160-bit ECC offers the comparable security to 1024-bit RSA. An elliptic curve over a finite field GF (a Galois Field of order q) is composed of a finite group of points (x_i, y_i), where integer coordinates x_i, y_i satisfy the long Weierstrass form:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \tag{1}$$

and the coefficients a_i are elements in $GF(q)$. Since the field $GF(q)$ (q is a prime) is generally used in cryptographic applications, (1) can be simplified to:

$$y^2 = x^3 + ax^2 + b, \tag{2}$$

where $a, b \in GF(q)$.

The elliptic curve points form an additive abelian group, so that the addition of any two points is a point in the group. Given two points P and Q, with the coordinates (x_1,y_1), (x_2,y_2), respectively, the addition results in a point R on the curve with coordinate (x_3,y_3), where x_3 and y_3 satisfy

$$(x_1,y_1) + (x_2,y_2) = (x_3,y_3), \tag{3}$$

such that

$$x_3 = L^2 + L + x_1 + x_2 + a, \tag{4}$$

$$y_3 = L(x_1 + x_3) + x_3 + y_1, \tag{5}$$

where

$$L = (y1 + y2)/(x1 + x2) \tag{6}$$

If $x_1 = x_2$ (note $x_1 + x_2$ is 0), then R is defined as a point at infinity, O. O is an identity element of the group. Each element in the group has an inverse that satisfies $P + (-P) = O$, and $(-P) + P = O$. Also, $P + O = O + P = P$. If $P = Q$, then $R = P + P = 2P$, and coordinate (x_3,y_3) is derived by

$$x_3 = L^2 + L + a, \tag{7}$$

$$y_3 = {x_1}^2 + (L+1)x_3, \tag{8}$$

where

$$L = x_1 + y_1/x_1. \tag{9}$$

The ECC relies on the difficulty of the Elliptic Curve Discrete Logarithm Problem, that is, given points P and Q in the group, it is hard to find a number k such that $Q = kP$.

2.2.2 Elliptic Curve Digital Signature Algorithm (ECDSA)

ECC signature is based on Digital Signature Algorithm. We assume Alice sends a message to Bob. To convince Bob that the message does come from Alice, Alice needs to apply a digital signature for the message so that Bob can verify it by using Alice's public key. Initially, Alice and Bob have to agree on a particular curve with base point P over the field $GF(p)$, and the order of P is q. When Alice sends a message to Bob, she attaches a digital signature (r,s) generated by following steps (suppose Alice has a private key x and a public key $Q = xP$).

1. Choose a random key k in $[1, q-1]$;
2. Compute kP, yield a point with coordinate (x_1,y_1). Let $r = x_1 \pmod{q}$. Check r, go back to the first step if the result is zero;
3. Compute $k^{-1} \pmod{q}$;
4. Compute $s = k^{-1}(Hash(m) + xr)$, where $Hash$ is a one-way hash function. Again, check s, go back to the first step if $s = 0$;
5. (r,s) is the digital signature.

To verify the message m and the signature, Bob needs to do following steps.

1. Compute $w = s^{-1} \bmod q$ and $H(m)$;
2. Compute $u_1 = H(m) \cdot w \bmod q$ and $u_2 = r \cdot w \bmod q$;
3. Compute $u_1 P + u_2 Q$, get the result point (x_2, y_2);
4. The signature is verified if $x_2 = r$.

Finally, Bob compares the value of x_2 and r, and accepts the message only if x_2 equals to r.

3 Implementation

We implement 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.

Given the limited processor resources, we concentrate most of our efforts on computation optimization. The fundamental RSA operation is large integer exponentiation over a finite field $GF(n)$, where n is the product of two large prime number p and q. The computation of the exponentiation can be decomposed to a series of squaring, multiplications and reductions. In addition, we also need an inversion module to calculate the public key and private key pair given two prime numbers p and q. In this section, we first present our optimization in large integer operations. Based on that, we describe our further optimization by using Montgomery reduction and Chinese Remainder Theorem (CRT) to significantly improve the computation efficiency. The fundamental ECC operation is large integer arithmetics over either prime number finite field $GF(p)$ or binary polynomial field $GF(2^m)$ (where m is a prime). Because the two heavily used operations: multiplication and modular reduction, can be more effectively optimized if pseudo-Mersenne primes are picked for elliptic curves compared with those of binary field [5], we limit our discussion in prime number finite field $GF(p)$ in this paper. Without further clarification, our discussion of ECC implementation is based on SECG recommended 160-bit elliptic curve: secp160r1.

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.

3.1 Large Integer Operations

We implement a suite of large integer arithmetic operations, including addition, subtraction, shift, multiplication, division and modular reduction. Due to the space limit, we only present three of most important functions: multiplication/squaring, modular reduction and inversion.

3.1.1 Multiplication and Squaring

The multiplication (or squaring) is the key component in RSA implementation because the exponentiation is basically computed by multiplications and squaring. We have compared three different multiplication implementations [5, 9, 8], and finally decided to use Hybrid Multiplication proposed in [5]. To ease our explanation, we use three large integers as the examples for our following discussion: $A(a_{n-1}, a_{n-2}, \cdots, a_1, a_0)$, $B(b_{n-1}, b_{n-2}, \cdots, b_1, b_0)$, and $C(n_{2n-1}, c_{2n-2}, \cdots, c_1, c_0)$, where $C = A * B$. A and B both have length of n words, each word has k-bit size. The product C has $2n$ words.

The Hybrid multiplication is the combination of Row-wise multiplication and Column-wise multiplication. The Row-wise method fixes the multiplier b_i ($0 \leq i \leq n$), and multiplies it with every word of multiplicand A. Partial results are stored in $n+1$ accumulator registers. Every time one row is finished, the last accumulator register can be stored to memory as the part of final results. On average, one memory load is required for each $k \times k$ multiplication. When integer size n is increased, the required number registers increase linearly in Row-wise method. For 1024-bit RSA, a typical multiplication is between two 128-byte large integers. Given only 32 registers in ATmega128, Row-wise multiplication can not be directly applied.

The Column-wise method, on the other side, computes the partial results of $a_i * b_j$ (where $i+j=l$) for column l. After one column finishes, the last word of accumulator registers is stored as the part of final result. The Column-wise method only requires three accumulator registers and two more for operands. However, two memory load operations are required for each $k \times k$ multiplication.

The Hybrid method takes advantages of Row-wise and Column-wise strategies. To optimize the memory operation, the Hybrid method merges a number (d) of columns together, and then conducts Row-wise multiplication in each merged column. When d equals to 1, the Hybrid method becomes the Column-wise multiplication. When d equals to n, then it becomes Row-wise method. A larger d leads to fewer memory operations, but requires more registers. A small d, however, requires more memory operations and consumes more CPU cycles. Balancing the advantages and disadvantages, we implement the Hybrid multiplication with column width $d=4$, which requires 9 accumulator registers, 5 operand registers, 6 pointer registers (point to A, B and C), and others for temporary storage and loop control.

We implement the Hybrid multiplication in assembly language. 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.

The 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%.

3.1.2 Modular Division

Modular division is another expensive operation in ECC. In Affine coordinate, each ECC operation of point addition and doubling requires a modular inversion. The integer inversion is also required for ECC digital signature generation and verification. In our implementation, we adopt the Great Divide scheme proposed in [12]. We briefly explain the algorithm in the followings.

Given an denominator x and numerator y, we want to compute the modular division $\frac{y}{x}$ over $GF(p)$. This is equivalent to find r, so that

$$r \equiv \frac{y}{x} (\text{mod q}). \tag{10}$$

To find r efficiently, the algorithm maintains following two invariant relationship:

$$A * y \equiv U * x, \text{ and } B * y \equiv V * x, \tag{11}$$

where A, B, U,and V are four auxiliary variables and initialized with values x, q, y, and 0, respectively. Note the two relationship is true with the initial values. The algorithm intuition is to reduce the value of A to 1, so that the first relationship in (11) will become $y \equiv U * x$, and U will be the result. The procedure is conducted in following way. When A is even, we can divide A by 2. Correspondingly, U has to be divided by 2 to keep the relation true. If U is not even at that time, we can make it become even by adding U with the modulus. When A is odd, we use the 2nd relationship to help to reduce A. If B is even, we keep dividing B by 2 similarly to make B odd. Then we add the two relation together and the divide the result value by 2 at the both sides. By repeating this process, it is guaranteed that either value of A or B reduces one bit in one iteration. The procedure stops when $A = B = 1$, the first equation becomes $y \equiv U * x$. The value of U is our final result. If we initialize U with 1, this routine can be used to calculate an inversion of x. This algorithm works when x and q are relatively prime. The Great Divide finishes division or inversion operation in $2(log(x) - 1)$ steps. Great Divide is much faster than the long division method because Great Divide only needs addition operations in each iteration, while long division method requires multiplications. Unfortunately, Great Divide cannot be used in RSA to calculate the public key and private key. The reason is that Great Divide only works when the modulus is an odd number, but the totient $\phi(n) = (p-1)(q-1)$ in RSA is always even. Therefore, we use Extended Euclidean algorithm instead.

3.1.3 Modular Reduction

The modular reduction operation is another important module because each multiplication or squaring must be followed by a reduction operation. The classic reduction method is using long division. Although the long division method is a general method for calculating the modular reduction, it is also the slowest method. In ECC cryptosystem, the modular reduction operation is as important as modular multiplication. Each multiplication must be followed by a reduction operation. Since 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. Because the optimization is curve specific, we will explain in more details in the section of ECC operation.

Now, we discuss the modular reductions in RSA and ECC digital signature generation and verification. In most cases, the modulus is not a pseudo-Mersenne prime, the optimization cannot be applied for those reduction calculation. 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. We briefly describe the long division method as in Algorithm 1. The long division producer reduces the remainder of x by one byte in each iteration.

3.1.4 Inversion

The multiplicative inversion is required to calculate the RSA public key and private key pair. A RSA public key e and a private key d should satisfy the condition: $e \times d \equiv 1 \bmod \phi(n)$, where totient $\phi(n) = (p-1)(q-1)$. Given a public key e, the corresponding private key is the multiplicative inversion of e. Since both p and q are prime numbers, $\phi(n)$ must be even. Thus, the efficient Great Divide scheme [12] can not be used because Great Divide requires the modulo to be an odd number. We use the classic Extended Euclidean Algorithm to compute the private key. The algorithm is described as below.

3.2 RSA Operations

With the basic large integer operation modules implemented, we conducted the first performance test for RSA public key operation (17-bit public key) and private key operation (1024-bit private key). Surprisingly, both operations are very slow. It takes 4.6s to finish the public key operation and 389s to do a private key operation.

Algorithm 1 Reduction by using long division.

```
1:  Input: $x, n$;
2:  Output: $r = x \bmod n$;
3:  **while** $x \geq n$ **do**
4:      Align the most significant byte (MSB) of modulus $n$ to the MSB of $x$, the lower bytes of $n$ can be
        filled with zeros;
5:      Starting with the MSB of $x$, divide the first two MSBs of $x$ by the MSB of modulus $n$, and get the
        quotient;
6:      Multiply the quotient with the modulus and get a subproduct;
7:      If the subproduct is greater than the remainder of $x$ (over estimation), subtract the modulus from the
        subproduct;
8:      Then subtract the subproduct from the remainder of $x$;
9:      The procedure continues and goes back to step 2 if the MSB of the remainder becomes zero;
10:     If the MSB of the remainder is not zero (under estimation), subtract the modulus from the remainder,
        and then go back to step 2;
11:     The procedure stops when the remainder is less than modulus $n$;
12: **end while**
13: return $x$;
```

To learn the reason for the poor performance of our initial implementation, we profiled the every operation in RSA exponentiation. We found that the modular reduction following each multiplication consumes 0.13s on the average. For 17-bit public key, there are totally 17 such reductions, which spend 2.2s in total, almost 50% of the execution time of the public key operation.

In this section, we explore two optimization schemes which aim to reduce the costs of the reduction and multiplication operations.

3.2.1 Montgomery Reduction

Montgomery reduction [10] 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} (mod 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 [10] for details. It seems this algorithm does not save anything because an extra step to convert TR^{-1} (mod N) to T modulo N is required. However, this method is useful if a number of computations are needed for the same modulus N. That is the reason that Montgomery reduction is widely used to reduce the reduction cost for the exponentiation operation in RSA. The efficient exponentiation by using Montgomery reduction is described as below. The idea is to convert integer b to an N-residue so that $b^a * 2^k$ (mod n) can be quickly computed without doing any reduction. As the result, we only need to do two reductions for the exponentiation. The first one is to convert b to N-residue before the Montgomery reduction, the second one is to convert the exponentiation result from N-residue back to integer. 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.

Algorithm 2 Extended Euclidean Algorithm

```
1:  Input: $e$, $\phi(n)$
2:  Output: $d$
3:  $x \leftarrow 0, lastx \leftarrow 1$
4:  $a \leftarrow e, b \leftarrow \phi(n)$
5:  **while** $b! = 0$ **do**
6:      $q \leftarrow a/b$
7:      $r \leftarrow a \bmod b$
8:      $a \leftarrow b$
9:      $b \leftarrow r$
10:     $temp \leftarrow lastx$
11:     $x \leftarrow lastx + q * x \bmod \phi(n)$
12:     $lastx \leftarrow temp$
13: **end while**
14: return $lastx$
```

Algorithm 3 An efficient exponentiation by using Montgomery reduction.

```
1:  Input: $b, a, n$
2:  Output: $c = b^a \pmod n$
3:  $c \leftarrow b \cdot 2^k \pmod n$, $(2^k > n)$
4:  $t \leftarrow c$
5:  **for** from $i = msb(a)$ to $i = 0$ **do**
6:      $c \leftarrow c^2 \cdot 2^{-k} \pmod n$
7:      **if** $i$'s bit of $a$ is set **then**
8:          $c \leftarrow c * t \cdot 2^{-k} \pmod n$
9:      **end if**
10: **end for**
11: $c \leftarrow c * 2^{-k} \pmod n$
12: return $c$
```

3.2.2 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. CRT can be found in any number theorem textbook, here we only give a simple example to serve for this paper. Let number n_1, n_2 be positive integers which are co-prime to each other, i.e., $GCD(n_1, n_2) = 1$. Let $n = n_1 * n_2$ and x_1, x_2 be integers. CRT states that if there are congruence: $x \equiv x_1 \pmod{n_1}$, $x \equiv x_2 \pmod{n_2}$, then there is only one solution x between 0 and $n-1$, inclusively. The value of x can be determined by

$$x = x_1 * r_1 * s_1 + x_2 * r2 * s_2 (\text{ mod n}), \tag{12}$$

where $r_i = \frac{n}{n_i}$, $s_i = r_i^{-1} \pmod{n_i}$ for $i = 1, 2$. Based on the above simple version of CRT, we describe our RSA optimization (adapted from [3]) by using CRT. Note step 3 and 4 can be precomputed. Th above algorithm reduces the size of a and d in half. Consider a and d are both 1024-bit integers, the computation of a^d is reduced to 2 modular exponentiation with both base and exponent size of 512 bits. Thus, the

Algorithm 4 Calculate a^d (mod n) by CRT

```
1: Input: $a, d, n, p, q$ $(n = p * q)$
2: Output: $m = a^d$ (mod $n$)
3: $R_p \leftarrow q^{p-1}$ (mod $n$), $R_q \leftarrow p^{q-1}$ (mod $n$)
4: $D_p \leftarrow d$ (mod $p-1$), $D_q \leftarrow d$ (mod $q-1$)
5: $A_p \leftarrow a$ (mod $p$), $A_q \leftarrow a$ (mod $q$)
6: $M_p \leftarrow A_p^{D_p}$ (mod $p$), $M_q \leftarrow A_q^{D_q}$ (mod $q$)
7: $S_p \leftarrow M_p * R_p$ (mod $n$), $S_q \leftarrow M_q * R_q$ (mod $n$)
8: $m = S_p + S_q$ (mod $n$)
9: return $m$
```

overall computational complexity is reduced to roughly 1/4 of the original exponentiation. The CRT can also be applied for public key operation, but the computational complexity can only be reduced by 50%. The reason is that public key size is normally very small (17 bit in our experiment), so the exponent size cannot be reduced in this case. 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.

3.3 ECC Operations

In this section, we present our optimization for ECC operation. We first discuss ECC point addition and doubling. We then introduce an optimized modular reduction for curve secp160r1. Finally, we explain several different optimizations for point multiplication.

3.3.1 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]. Consider an ECC point in Modified Jacobian coordinate, $P_1(X_1, Y_1, Z_1, aZ_1^4)$, and a point in Affine coordinate, $P_2(x_2, y_2)$, their addition results in the third point $P_3 = (X_3, Y_3, Z_3, aZ_3^4)$ in Modified Jacobian coordinate. The result is given by following equations.

$$\begin{aligned} X_3 &= -H^3 - 2X_1H^2 + r^2, \\ Y_3 &= -Y_1H^3 + r(X_1H^2 - X_3), \\ Z_3 &= Z_1H, \\ aZ_3^4 &= aZ_3^4, \end{aligned} \tag{13}$$

where $H = x_2Z_1^2 - X_1$, and $r = y_2Z_1^3 - Y_1$. The result of point doubling for $P_3 = 2P_1$ is given by following formula.

$$\begin{aligned} X_3 &= T, \\ Y_3 &= M(S-T) - U, \\ Z_3 &= 2Y_1Z_1, \\ aZ_3 &= 2U(aZ_1^4) \end{aligned} \tag{14}$$

To estimate the computational complexity, we only consider large integer multiplication and squaring operations, and ignore those addition and subtraction since they are much faster. According to Eq.13 and Eq.14, point addition requires 9 large integer multiplications and 5 squaring, and point doubling requires 4 multiplications and 5 squaring.

The basic point operations can be further optimized for specific elliptic curves. In our case, the curve parameter a of secp160r1 equals to -3. For point doubling, M can be further reduced to

$$M = 3X_1^3 - 3Z_1^4 = 3(X_1 + Z_1^2)(X_1 - Z_1^2). \tag{15}$$

As the result, point doubling operation reduces to 4 multiplications and 4 squaring. Actually, aZ_3^4 does not have to be calculated in point addition, so the computational complexity reduces to 8 multiplications and 3 squaring. Our observation supports the choice of mixed coordinate, the performance of point multiplication improves around 6% compared with our previous implementation in Jacobian coordinate.

3.3.2 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. In this section, we use curve secp160r1 as the example to show how to do efficient reduction.

Suppose we use the 8-bit architecture, the multiplication result of two 160-bit integers can be represented by

$$C(c_{39}, \cdots, c_{20}, c_{19}, \cdots, c_1, c_0),$$

where c_i $(0 \le i \le 39)$ is a word with 8 bits, and c_{39} is the most significant word. The 40-word integer can also be written as:

$$C = (c_{39}, \cdots, c_{20}) * 2^{160} + (c_{19}, \cdots, c_1, c_0) \tag{16}$$

Given the field of curve secp160r1 $q = 2^{160} - 2^{31} - 1$, we can have $2^{160} \equiv 2^{31} + 1$. Therefore,

$$\begin{aligned} C &\equiv (c_{39}, \cdots, c_{20}) * (2^{31} + 1) + (c_{19}, \cdots, c_1, c_0) \\ &\equiv (c_{39}, \cdots, c_{20}) * 2^{31} + (c_{39}, \cdots, c_{20}) + (c_{19}, \cdots, c_1, c_0) \end{aligned} \tag{17}$$

Since each word has 8 bits, the first term in the result of Eq. 17 can be further reduced to

$$\begin{aligned} &(c_{39}, \cdots, c_{20}) * 2^{31} \equiv (c_{39}, c_{38}, c_{37}) * 2^{167} + c_{36} * 2^{159} + (c_{35}, \cdots, c_{20}) * 2^{31} \\ &\equiv (c_{39}, c_{38}, c_{37}) * 2^{38} + (c_{39}, c_{38}, c_{37}) * 2^{7} + (d_7d_6 \cdots d_0) * 2^{31} + (d_7d_6 \cdots d_0) + (d_0) * 2^{159} \end{aligned} \tag{18}$$

where $(d_7, \cdots, d_1, d_0)$ are 8 bits of c_{36}. Now, all terms in Eq.17 and 18 have at most 159 bit length, the reduction result is simply the addition of these terms.

3.3.3 Further Optimization

Examining the computational complexity, we notice that point addition is more expensive than point doubling. As we have discussed, point multiplication can be decomposed to a series of point addition and doubling, we would rather use more point doubling than point addition to compute the point multiplication. Morain *et al.* found Non-adjacent forms (NAFs) is an effective way to achieve the lightest Hamming weight for scalar k in point multiplication $k*P$, which results to use the least number of point additions to calculate $k*P$ [11]. For example, $255*P$, or $(11111111)*P$, requires 7 point additions. But if we transform it to $(10000000-1)*P$, which is $256*P-P$, only one addition is required. Note the point subtraction can be replaced by point addition because the inverse of an Affine point $P=(x,y)$ is $-P=(x,-y)$. We implement NAFs technique in random point multiplication. According to our experiments, point multiplication with NAFs contributes at least 5% performance improvement.

Recall in the digital signature procedure in ECDSA, component r is generated by a point multiplication with the fixed base point of a selected elliptic curve. To further reduce the execution time, we precompute some partial results and apply sliding window method [7] to speed up fixed point multiplication. Different from NAFs, sliding window scheme groups scalar k into a number of $s-bit$ bit-clusters, where s is also called window size. So, k can be represented by $k_m*2^{sm}+k_{m-1}*2^{s(m-1)}+\cdots+k_0$, where k_i is a bit-cluster. If we precompute the point multiplication with every possible value of k_i, the number of point addition is bounded by $\lceil\frac{160}{s}\rceil-1$. Note the sliding window method does not reduce the number of point doubling operations. Obviously, this scheme requires extra memory space for storing partial results. In practice, 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.

Our initial experimental results indicated that it took double amount of time to perform an ECDSA verification than to do an ECDSA signature: signature is 1.35s, while verification is 2.85s. The reason is that the verification requires two ECC point multiplications (while the signature only needs one point multiplication); the verifier has to perform u_1P+u_2Q as shown in Section 2.2.2. To speed up the verification time, we adopt Shamir's trick [6] to do multiple point multiplication simultaneously. The idea of Shamir's trick is similar to the sliding window method discussed previously. Given t-bit u_1 and u_2, we use the window size ω and precompute the values $iP+jQ$ for $0\leq i,j\leq 2^{\omega}$. At each of $\lceil t/\omega\rceil$ steps, we perform ω doubling and the (precomputed) additions determined by the window contents. The larger the window size (ω) is, the more memory is required for storing the precomputed values. In practice, we choose the single bit window size, $\omega=1$. Therefore, only the value of $P+Q$ needs to be precomputed and stored. As the result, the performance of ECDSA verification has been improved more than 30%, from 2.85s to 1.96s. There is still further improvement space if multi-bit window size is used, but the trade-off is more memory overhead.

4 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. MICAz is powered by ATmega128 microcontroller. ATmega incorporates an 8MHz, 8-bit RISC CPU, 128K bytes flash memory (ROM) and 4K RAM. The RF transceiver on MICAz is IEEE 802.15.4/ZigBee compliant, and can have 250kbps data rate. 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 1.35s to generate a signature and 1.96s 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.

4.1 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.

4.1.1 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 [5], 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 to implement 1024-bit RSA on MICAz motes is the memory constraint. MICAz mote only has 4KB RAM, which is the total space can be used by 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 module 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. Actually we do not need this module in the real time. This helps us a lot because it consumes almost 1KB temporary space.

4.1.2 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. A ”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.

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

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

Table 1: Execution time profiles of some important modules.

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 20.5s, closely matching our experiment result of 21.5s.

4.2 ECC Evaluation

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

4.2.1 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[7] 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 individually. 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 2 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.

	FPM	RPM	PAdd	PDbl	SIGN	VERIFY
ECC	1.24s	1.35s	6.33ms	5.87ms	1.35s	1.96s

Table 2: Execution Time of ECC point 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.

Next, we implement ECDSA signature scheme. The experimental results are shown in Table 2. In fact, when signing a message, one fixed point multiplication is the dominant operation. 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 3 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.

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

Table 3: ECC implementation code size.

4.2.2 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.

4.3 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 [5] and give the possible explanation of the performance gap.

	FPM	RPM	PAdd	PDbl	SIGN	VERIFY
MICAz	1.24s	1.35s	6.2ms	5.8ms	1.35s	1.96s
TelosB	1.44s	1.55s	7.3ms	7.0ms	1.55s	2.25s

Table 4: 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.

To learn the performance of the public key cryptosystems on different sensor platforms, we have revamped our previous ECC implementation on TelosB mote[13]. We summarize the performance comparison in Table 4. 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 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 [5]. 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.

5 Conclusion

In this technical report, 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 MICAz 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 1.35s and 1.96s respective for Mica motes, and 1.55s and 2.25s 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.

Acknowledgment This project was partially supported by US National Science Foundation award CCF-0514985.

References

[1] H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation. In *Advances in Crytology-Proceedings of ICICS, Lecture Notes in Computer Science*, pages 282–290, Springer-Verlag, 1997.

[2] H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. In *ASIACRYPT: Advances in Cryptology*, 1998.

[3] J. Grobchadl. The Chinese Remainder Theorem and its application in a high-speed RSA crypto chip. In *ACSAC*, page 384, 2000.

[4] Vipul Gupta, Matthew Millard, Stephen Fung, Yu Zhu, Nils Gura, Hans Eberle, and Sheueling Chang Shantz. Sizzle: A Standards-based end-to-end Security Architecture for the Embedded Internet. In *Third IEEE International Conference on Pervasive Computing and Communication*, Kauai, Mar. 2005.

[5] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. In *CHES*, Boston, Aug. 2004.

[6] D. Hankerson, A. J. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, 2004.

[7] C. K. Koc. High-Speed RSA Implementation. In *RSA Laboratories TR201*, Nov 1994.

[8] An Liu and Peng Ning. TinyECC: Elliptic Curve Cryptography for Sensor Networks. Sept 15 2005.

[9] D.J. Malan, M. Welsh, and M.D. Smith. A public-key infrastructure for key distribution in tinyos based on elliptic curve cryptography. In *The First IEEE International Conference on Sensor and Ad Hoc Communications and Networks*, Santa Clara, CA, October 2004.

[10] P. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Communication*, 44(170):519–521, April 1985.

[11] 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.

[12] S. Chang Shantz. From Euclid's GCD to Montgomery Multiplication to the Great Divide. In *Technical report, Sun Microsystems Laboratories TR-2001-95*, June 2001.

[13] Haodong Wang, Bo Sheng, and Qun Li. Elliptic curve cryptography based access control in sensor networks. *International Journal of Sensor Networks*, 1(2), 2006.