# RSA Implementation Documentation

Samuel Grant Dawson Williams

August 19, 2010

## Contents

## 1 Introduction

This documentation outlines the main functions and structure of the given RSA implementation. Several algorithms were devised by my own idea such as division by repeated doubling. The implementation's performance is then tested and discussed.

## 2 Compile and Run

To compile and run, use the following commands. Basic generation algorithm reads text from file **text.txt**. Press any key when program says 'Waiting...':

```
$ g++ -o multiple multiple.cpp -O3
$ ./multiple
Generating key A
.
.
..............................................
p = 0x3928CE8A98673B3F1C782A4620EC570B651FA885FBA61535
q = 0x9D2A06A4288E244213494A6B588F3DE648E34DCCB1FB16B9
n = 0x231768D87E3DFDA230E599007A6F4ED1D1A477246FC4653A \
      A0DD0B5EE66724394EA8CDEF46BC0BCADD70B42F90CEE14D
.
e = 0x43827EB99DF7B861C132A8B122D4DA7288827AA9A5EAEAA3
d = 0x04A86B7DDC1AB9CB75135C5DC555847DCE9D1727557739F7 \
      D89F84C6DA91C5AB603D92637D2EFDA81D68B8A0F3424ECB
Generating key B
................................
.......................................................
p = 0xC6E6FDD882B65BA6C18DC6884CB3AD10CD88C3F92513848F
q = 0x6D19AEFBE16704A9325C01A5FC13B3F899195D1F50100ED3
n = 0x54C44E98ADF1849F8D5DE63384019EFF257BA00755901953 \
      09D06AB18E7DF25DDE4DEB6AE54A142DFB159156994613DD
..........
e = 0x53546D1448661B0133B59A61241B7451AD76D628708F2715
d = 0x33CF0AE12EAF73DFB927B7497B561655C1FD75F19B4F9E42 \
      A3513755CE6FD2BAF27974ED40A588674465ED1E755E1535
Time for key generation: 1.95216s
Waiting...

Packing from 12 to 11
     Encrypted Text: {
0x2C5B71D62E9D26F30423F3BDEC77DA0028FE0ED8BD6328FA \
  BBDAE97C214491C77B0F474324AFEC5A7A0029472ECA229C,
0x35E3028BF1C335D4A8C250648D26D52A361020CA6B317F59 \
  9E096934C8E9849229E3A4277F289D320F385F8E3C4D99F0,
0x28E2ED31931252881BC504E65F920F1F9B0C33CCFB4EED7B \
  1759826657D11E7BD819E22D9CAEB302C693F19237911EB8
}
Time for encryption: 0.0666592s
Waiting...

Packing from 11 to 12
     Decrypted Text: The quick brown fox jumped over the lazy dog.
Time for decryption: 0.078691s
         Total time: 2.09751
   Pack/Unpack time: 0.000105619s
          Processed: 1238.39 bytes per second
```

# 3 Implementation

## 3.1 Basic Mathematics

Random numbers are generating using cryptographically secure **/dev/random**. Length is specified in multiple of 32-bit words.

Listing 1: Random Number Generation

```
// Returns a large random number. Digits is in multiple of 32 bits.
void Integer::generateRandomNumber (Integer::DigitT length) {
    static std::ifstream * randomDevice = NULL;

    if (!randomDevice) {
        randomDevice = new std::ifstream("/dev/urandom", std::ios::binary);
    }

    m_value.resize(length);
    randomDevice->read((char*)&m_value[0], (std::size_t)m_value.size() * sizeof(DigitT));
}
```

Greatest common divisor is calculated using non-recursive function. This function requires one division per iteration.

Listing 2: Greatest Common Divisor

```
// Returns the greatest common divisor of a and b.
void Integer::calculateGreatestCommonDivisor (Integer a, Integer b) {
    while (b != 0) {
        Integer tmp = a;
        tmp.modulus(b);

        a = b;
        b = tmp;
    }

    (*this) = a;
}
```

Powers are calculated using repeated squaring. This operation is inefficient for large numbers. Performance is $O(k)$ for k bit exponent. Each iteration performs at most two multiplications.

Listing 3: Calculate Power

```
void Integer::setPower (Integer base, Integer exponent) {
    (*this) = 1;

    while (exponent != 0) {
        if (exponent.m_value[0] & 1) {
            this->multiply(base); // % m
        }

        exponent.shiftRight(1);
        base.multiply(base); // % m
    }
}
```

## 3.2 Barrett Modular Reduction

Barrett modular reduction is used to optimise in the case we are calculating $n \bmod m$. It works by removing the need to use division to calculate modulus, which in many cases means that division can be removed completely from the inner loop of an algorithm. It is very versatile and can be used in several places where $n \bmod m$ is calculated with constant $m$ several times.

Listing 4: Barrett Reduction

```
BarrettReduction::BarrettReduction (const Integer & _mod) {
    mod = _mod;
    mod.normalize();

    // Remainder - temporary.
    Integer r;

    // Radix - the number of possible values per digit
    b = (Integer::IntermediateT)1 << Integer::DIGIT_BITS;
    bk.setPower(b, mod.size() * 2);
    mu.setFraction(bk, mod, r);

    // Division by 0x100 is the same as shiftRight(2)
    bn = Integer::DIGIT_BITS * (mod.size() - 1);
    bp = Integer::DIGIT_BITS * (mod.size() + 1);

    // Mask for base-2 modulus
    bkp.setPower(b, mod.size() + 1);
    bkm = bkp;
    bkm.subtract(1);
}

void BarrettReduction::modulus (Integer & x) const {
    Integer q1, q2, q3, r1, r2;

    q1 = x;
    q1.shiftRight(bn);

    q2.setProduct(q1, mu);

    q3 = q2;
    q3.shiftRight(bp);

    r1 = x;
    r1.binaryAnd(bkm);

    r2.setProduct(q3, mod);
    r2.binaryAnd(bkm);

    if (r2 > r1) {
        r1.add(bkp);
    }

    r1.subtract(r2);

    while (r1 >= mod) {
        r1.subtract(mod);
    }

    x = r1;
    x.normalize();
}
```

## 3.3 Modular Exponentiation

Barrett modular reduction can improve performance of $x^y$ mod $m$, because we change division to multiplication in the inner loop.

Listing 5: Calculate Power using Barrett Modular Reduction

```
void Integer::setPower (Integer base, Integer exponent, const BarrettReduction & r) {
    (*this) = 1;

    while (exponent != 0) {
        if (exponent.m_value[0] & 1) {
            this->multiply(base);
            r.modulus(*this);
        }

        exponent.shiftRight(1);
        base.multiply(base);
        r.modulus(base);
    }
}
```

Jacobi test is calculated using non-recursive function. This function will tell us if $b$ is composite or not. We perform one division per loop.

Listing 6: Jacobi Test

```
int jacobi (Integer m, Integer n) {
    int i = 1;
    Integer t;

    while (m > 1) {
        Integer j = 0;

        while ((m[0] & 1) == 0) {
            j.add(1);
            m.shiftRight(1);
        }

        if ((j[0] & 1) == 1) {
            t = n[0] & 7;

            if ((t == 3) || (t == 5))
                i = -i;
        }

        if ((m[0] & 3) == 3 && (n[0] & 3) == 3) {
            i = -i;
        }

        t = n;

        t.modulus(m);
        n = m;
        m = t;
    }

    return i;
}
```

## 3.4 Probabilistic Prime Generation

To test prime number $p$ first we generate a random number $a$ in the range $(2, p)$. We check that it is co-prime to $b$ using $GCD$ test. If the number is co-prime, we use $Jacobi$ function and test equivalency with $Legendre$ function. We repeat this test several times to increase probability of accurate result.

Listing 7: Probabilistic Prime Test

```
bool Integer::isProbablyPrime (int tests) const {
    const Integer & p = *this;

    if (p == 2) {
        return true;
    }

    // Cache the reduction for better setPower.
    BarrettReduction br (p);

    while (tests-- > 0) {
        Integer a = 0;
        a.generateRandomNumber(2, p);

        Integer gcd = 0;
        gcd.calculateGreatestCommonDivisor(a, p);

        if (gcd == 1) {
            Integer l = 0, e = 0, p1 = p;
            p1.subtract(1);
            e = p1;
            e.shiftRight(1);

            l.setPower(a, e, br);

            int j = jacobi(a, p);

            if (((j == -1) && (l == p1)) || ((j == 1) && (l == 1))) {
                // So far so good...
            } else {
                // p is composite
                return false;
            }
        } else {
            // p is composite
            return false;
        }
    }

    return true;
}
```

To generate a prime, we use random number generation to select numbers. We ensure that we are only checking odd numbers, and then use probabilistic method to determine if the number is likely to be prime. We also employ a small check to ensure we avoid Mersenne primes.

Listing 8: Prime Number Generation

```
void Integer::generatePrime (DigitT length) {
    while (true) {
        std::cout << "." << std::flush;
        this->generateRandomNumber(length);
        this->m_value[0] |= 1; // Ensure odd number

        if (isProbablyPrime()) {
            std::cout << std::endl;
            return;
        }
    }
}
```

## 3.5 Multiple Precision Mathematics

All operations function using 32-bit word as the default storage. This implies that we are using base $2^32$ number system. Operation has no maximum size, but performance will decrease as number of words used to represent number increases.

Addition is calculated by using double-word size arithmetic $(cr) = a + b$ where $c$ is carry and $r$ is the result for this word. Performance is $O(max(k, j))$ for adding $k$ digits and $j$ digits numbers together.

Listing 9: Multiple Precision Addition

```
void Integer::add(const Integer & a) {
    if (m_value.size() < a.m_value.size()) {
        m_value.resize(a.m_value.size());
    }

    IntermediateT carry = 0;
    std::size_t i = 0;

    for (; i < a.m_value.size(); i += 1) {
        IntermediateT result = (IntermediateT)m_value[i] +
                               (IntermediateT)a.m_value[i] + carry;

        m_value[i] = (DigitT)result;
        carry = result >> DIGIT_BITS;
    }

    for (;carry != 0 && i < m_value.size(); i += 1) {
        IntermediateT result = (IntermediateT)m_value[i] + carry;

        m_value[i] = (DigitT)result;
        carry = result >> DIGIT_BITS;
    }

    if (carry != 0) {
        m_value.push_back(carry);
    }
}
```

Subtraction is calculated word by word using double-word size to accumulate the total amount to be subtracted. We compare this with the amount we have, and if it is less, we can subtract. If it is not enough, we need to borrow from the right hand side. Upper bound for performance is $O(k)$ for subtracting $j$ digits from $k$ digits numbers together, where $k \geq j$.

Listing 10: Multiple Precision Subtraction

```
void Integer::subtract(const Integer & _a) {
    Integer a = _a; a.normalize();

    std::size_t width = a.m_value.size();
    IntermediateT take = 0;

    for (std::size_t i = 0; i < width; i += 1) {
        IntermediateT remove = take;

        if (i < a.m_value.size())
            remove += (IntermediateT)a.m_value[i];

        if (m_value[i] >= remove) {
            m_value[i] -= remove;

            take = 0;
        } else {
            width = std::max(width, i+2);

            m_value[i] = ((IntermediateT)m_value[i] + B) - remove;

            // Take 1 from the next digit
            take = 1;
        }
    }
}
```

Multiplication is calculated by using double-word size arithmetic $(cr) = a * b$ where $c$ is carry and $r$ is the result for this word. We keep track of current result, and accumulate the inner product and carry at each step for each digit. Performance is $O(kj)$ where we are multiplying two numbers of $k$ and $j$ digits.

Listing 11: Multiple Precision Multiplication

```
void Integer::setProduct(const Integer & x, const Integer & y) {
    assert(x.size() != 0 && y.size() != 0);

    std::size_t n = x.size() - 1, t = y.size() - 1;

    for (std::size_t i = 0; i < m_value.size(); i++)
        m_value[i] = 0;

    m_value.resize(x.size() + y.size() + 2);

    for (std::size_t i = 0; i <= t; i += 1) {
        IntermediateT carry = 0;

        for (std::size_t j = 0; j <= n; j++) {
            IntermediateT product = (IntermediateT)x[j] * (IntermediateT)y[i];
            IntermediateT result = (IntermediateT)m_value[i+j] + product + carry;

            m_value[i+j] = (DigitT)result;
            carry = result >> DIGIT_BITS;
        }

        m_value[i+n+1] = carry;
    }

    this->normalize();
}
```

Division is calculated by using double-word and quad-word size arithmetic (implemented using another multiple precision integer). We perform some basic checks such as whether division will yield a non-zero quotient, and then we normalise the number.

Listing 12: Multiple Precision Normalisation for Division

```
void Integer::setFraction(const Integer & numerator,
                          const Integer & denominator,
                          Integer & remainder)
{
    if (numerator < denominator) {
        (*this) = 0;
        remainder = numerator;
        return;
    }

    Integer x = numerator;
    Integer y = denominator;
    x.normalize();
    y.normalize();

    // Normalize?
    std::size_t shift = 0;

    DigitT back = y.m_value.back();
    while (back < (B/2)) {
        back = back << 1;
        shift += 1;

        assert(back != 0);
    }

    if (shift) {
        x.shiftLeft(shift);
        y.shiftLeft(shift);
    }

    setFraction(x, y);

    if (shift) {
        x.shiftRight(shift);
    }

    remainder = x;
}
```

Division itself is performed by looking at the first two digits in the denominator and estimating the digit for the quotient. We refine this guess using three digits from the denominator. We then subtract the $quotient \times denominator$, and continue to calculating next value for the quotient. Upper bound on performance is $O(kj)$ where we are dividing a number of $k$ digits by $j$ digits. However actual performance is better, specifically we can say that there will be at most $k - j$ single-precision divisions when normalisation is used.

Listing 13: Multiple Precision Division

```
void Integer::setFraction(Integer & x, Integer y) {
    const std::size_t n = x.size() - 1, t = y.size() - 1;

    if (n < t) return; // numerator is smaller than denominator, done.

    const std::size_t nt = n - t;

    Integer & q = *this;
    q.m_value.clear();
    q.m_value.resize(nt+1);

    Integer bb, bp, tmp1, tmp2;
    bb.setProduct(B, B); // Base (radix)
    bp.setPower(B, nt);

    if (y[t] < (B/2)) {
        std::cerr << "Thar␣be␣the␣dragons!" << std::endl;
    }

    tmp1.setProduct(y, bp);
    while (x >= tmp1) {
        q[nt] += 1;
        x.subtract(tmp1);
    }

    for (std::size_t i = n; i > t; i--) {
        if (x[i] == y[t]) {
            q[i-t-1] = B - 1;
        } else {
            IntermediateT t1 = (IntermediateT)x[i] * B;
            if (i > 0) t1 += x[i-1];
            q[i-t-1] = t1 / (IntermediateT)y[t];
        }

        // If we had 128 bit arithmetic, we could do this on the CPU.
        Integer u;
        u.setProduct(y[t], B);
        if (t > 0) u.add(y[t-1]);

        Integer v;
        v.setProduct(x[i], bb);
        if (i > 0) { tmp1.setProduct(x[i-1], B); v.add(tmp1); }
        if (i > 1) v.add(x[i-2]);

        while (true) {
            tmp1.setProduct(q[i-t-1], u);
            if (tmp1 > v) {
                q[i-t-1] -= 1;
            } else {
                break;
            }
        }

        tmp1.setPower(B, i-t-1);
        tmp2.setProduct(y, tmp1);
        tmp1.setProduct(q[i-t-1], tmp2);

        if (tmp1 > x) {
            x.add(tmp2);
            x.subtract(tmp1);
            q[i-t-1] -= 1;
        } else {
            x.subtract(tmp1);
        }
    }
}
```

Division can also be calculated by using repeated doubling. Performance of this algorithm might be as good as $O(kj \log j)$. However, in practice while there it has slightly different characteristics than the traditional algorithm given previously. Firstly, if the numerator and denominator are far apart, the staircase upwards can be very high. This is $O(logj)$, then we must descend, however this operation is probably $O(j)$, because during descent we sometimes ascend again. If descent can be improved to $O(\log j)$, performance of the algorithm may become $O(k \log j)$.

Listing 14: Multiple Precision Division

```
bool Integer::setFractionSlow(const Integer & numerator,
                              const Integer & denominator,
                              Integer & remainder)
{
    if (numerator == 0) {
        remainder = 0;
        (*this) = 0;

        return true; // no remainder
    }

    if (denominator == 0) {
        throw std::runtime_error("Division␣by␣0!");
    }

    (*this) = 0; // Number of divisions possible.
    Integer accumulator = 0; // Total value of doublings.

    // This pair must typically be shifted together.
    Integer count = 1;
    Integer product = denominator;

    while (true) {
        //std::cout << count << std::endl;

        Integer tmp = product;
        tmp.add(accumulator);

        int s = tmp.compareWith(numerator);

        if (s == 0) {
            // We have found a division with no remainder
            this->add(count);
            //accumulator.add(product);
            remainder = 0;

            return true;
        } else if (s == 1) {
            // We have found a division with a remainder
            count.shiftRight(1);
            product.shiftRight(1);

            if (count == 0) {
                // We have found a divisor with a remainder
                remainder = numerator;
                remainder.subtract(accumulator);

                return false;
            }

            continue;
        }

        this->add(count);
        accumulator.add(product);

        count.shiftLeft(1);
        product.shiftLeft(1);
    }
}
```

Other operations such as shifts, equality, comparison and logical operators are trivial, and can be reviewed in the source code **Integer.cpp**. Performance of these basic operations is generally linear based on size of input. Also base-16 input and output were implemented for debugging purposes.

## 3.6 Message Packing

We pack message according to size of $s$ determined by key's $n$ size.

Listing 15: Message Pack

```
TextT pack(StringT input, std::size_t s)
{
    const std::size_t PACK_BLOCKS = s;
    const std::size_t PACK_BYTES = sizeof(Integer::DigitT) * PACK_BLOCKS;

    TextT output;
    input.resize(((input.size() + PACK_BYTES - 1) / PACK_BYTES) * PACK_BYTES);

    for (std::size_t i = 0; i < input.size(); i += PACK_BYTES) {
        Integer j((Integer::DigitT*)&input[i], PACK_BLOCKS);

        output.push_back(j);
    }

    return output;
}
```

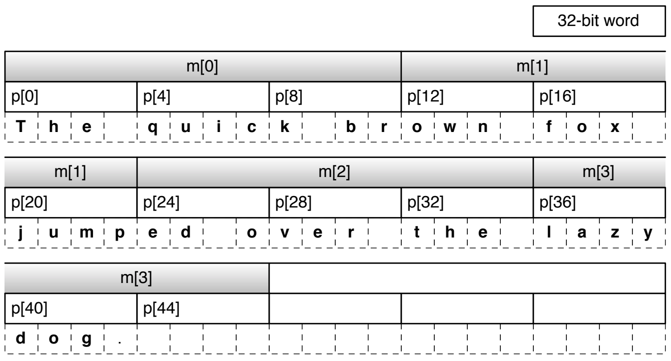

Figure 1: An example of packing plaintext $p$ into message $m$ with size 3.

A function is also provided to unpack the message data into a byte string. It works almost exactly the same way, but in reverse.

Listing 16: Message Pack

```
StringT unpack(const TextT & input, std::size_t s)
{
    const std::size_t PACK_BLOCKS = s;
    const std::size_t PACK_BYTES = sizeof(Integer::DigitT) * PACK_BLOCKS;

    StringT output;
    output.resize(input.size() * sizeof(Integer::DigitT) * PACK_BLOCKS);

    for (std::size_t i = 0; i < output.size(); i += PACK_BYTES) {
        input[i / PACK_BYTES].unpack((Integer::DigitT*)&output[i], PACK_BLOCKS);
    }

    return output;
}
```

A function is used to convert data from one packing size to another. It simply unpacks the data from the original size and packs it again into the new size.

Listing 17: Message Pack

```
TextT repack(const TextT & input, std::size_t s1, std::size_t s2)
{
    StringT buffer = unpack(input, s1);
    return pack(buffer, s2);
}
```

### 3.6.1 Calculating $s$

The argument $s$ is calculated according to the maximum size of the data during pack and unpack.

For packing, $s$ indicates that the data is segmented into blocks of $s \times 4$ bytes (see figure 1). The size of $m$ must be one word size less than the modulus, or we risk the possibility that $m >= n$.

For unpacking, we need to specify $s$ such that $s$ is bigger than the maximum size of the integer in 32-bit words (see figure 2). In the case of repacking encrypted data, the maximum size is the same as the size of the encryption keys $n$, since the result of $m^p \bmod n$ is always smaller than $n$.

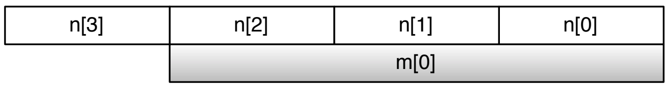

Figure 2: We choose $n$ for packing by looking at the size of $n$ in $m^p \bmod n$

### 3.6.2 Padding and Cryptographic Integrity

Block padding can have serious implications to the cryptographic integrity of message transmission. If padding at the end of a message is predictable, it may be easier to extract details about the encryption and decryption key. The above padding scheme does not take this into account.

## 3.7 Key Generation, Encryption and Decryption

To calculate decryption key, we need a function to compute inverse in $Z_v^*$. This function uses an iterative algorithm as recommended by Knuth. Some adaptation from the original algorithm have been performed reduce the amount of work being performed.

Listing 18: $Z_v^*$ Inverse Computation

```
//  Computes inv = u^(-1) mod v */
//  Reference: Knuth Algorithm X Vol 2 p 342
void Integer::calculateInverse (Integer u, Integer v) {
    Integer u1 = 1, u3 = u, v1 = 0, v3 = v;
    bool odd = false;

    Integer q, t3, w, t1;
    while (v3 != 0) {
        q.setFraction(u3, v3, t3);

        w.setProduct(q, v1);

        t1 = u1;
        t1.add(w);

        u1 = v1;
        v1 = t1;
        u3 = v3;
        v3 = t3;

        odd = !odd;
    }

    if (odd) {
        (*this) = v;
        this->subtract(u1);
    } else {
        (*this) = u1;
    }
}
```

Key generation is done by generating prime numbers. We define **bits** to be the key size in bits, which is multiple of 32.

Listing 19: Calculate $p$ and $q$

```
    // Find two large primes using probabilistic method.
    p.generatePrime(bits / Integer::DIGIT_BITS);

    do {
        q.generatePrime(bits / Integer::DIGIT_BITS);
    } while (q == p);
```

We then generate encryption key and decryption key.

Listing 20: Calculate $e$ and $d$

```
    Integer n;
    n.setProduct(p, q);
    Integer e = 0;

    // Generate public key
    e.generatePrime(bits / Integer::DIGIT_BITS);

    Integer p1 = p, q1 = q;
    p1.subtract(1);
    q1.subtract(1);

    // Generate private key
    std::cerr << "Generating␣d..." << std::endl;
    Integer phi = 0;
    phi.setProduct(p1, q1);
    Integer d = 0;
    d.calculateInverse(e, phi);
```

Once we have keys, actual encryption and decryption is easy, provided by a single funtion **transformMessage**. This also utilises Barrett modular reduction to enhance speed.

Listing 21: Transform Message

```
typedef std::vector<Integer> TextT;
TextT transformMessage(Integer e, Integer n, TextT message) {
    TextT result;

    BarrettReduction br(n);
    Integer c;

    for (TextT::iterator i = message.begin(); i != message.end(); i++) {
        c = *i;

        c.setPower(*i, e, br);

        result.push_back(c);
    }

    return result;
}
```

## 3.8 Secure Authentication

Secure authentication is performed by calculating $E_a(D_b(E_b(D_a(M)))) = M$. Both Alice and Bob have generated RSA keys.

Listing 22: Authentication Method

```
void testEncryption (StringT input, std::size_t bits) {
    // Alice's public and private key
    RSAKeys keysA = generateKeyPair(bits);

    // Bob's public and private key
    RSAKeys keysB = generateKeyPair(bits);

    // Alice will transmit input to Bob, securely
    TextT cipherText = pack(input, keysA.n.size() - 1);

    // Alice knows Bob's public key pair, and her own keys.
    // Firstly, Alice encrypts the text using her private key.
    cipherText = transformMessage(keysA.d, keysA.n, cipherText);
    cipherText = repack(cipherText, keysA.n.size(), keysB.n.size() - 1);
    // She then encrypts the data with Bob's public key.
    cipherText = transformMessage(keysB.e, keysB.n, cipherText);

    // Transmit from Alice to Bob
    TextT decipherText = cipherText;

    // Bob knows Alice's public key pair, and his keys.
    // Next, Bob decrypts the message using his private key.
    decipherText = transformMessage(keysB.d, keysB.n, decipherText);
    decipherText = repack(decipherText, keysB.n.size() - 1, keysA.n.size());
    // Finally, Bob decrypts the message using Alice's public key.
    decipherText = transformMessage(keysA.e, keysA.n, decipherText);

    // Original message is recovered.
    StringT output = unpack(decipherText, keysA.n.size());
}
```

Firstly we pack plaintext into cipher text using $s$ dictated by Alice's key. We then transform the cipher text $C_1 = D_a(M)$. We then repack the cipher text using $s$ dictated by Bob's key. We then perform $C_2 = E_b(C_1)$. Afterwards, we have cipher text which can transfer across the network securely.

Bob receives the message and decrypts it using his private key $C_3 = D_b(C_2)$. Finally, he repacks appropriately for Alice's key, and retrieves the message by calculating $M = E_a(C_3)$.

# 4 Performance

## 4.1 Key Generation

Performance is acceptable even for key generation and encryption up to 1024 bits. After this point, exponential curve begins to cause a large increase in run time. Barrett modular reduction is critical for generating larger keys, as without it the time required increases much more rapidly.

Key generation tests were repeated 30 times for k-bit keys where k is a multiple of 128 up to 2048. The tests were performed on a duel core CPU running on 2.5Ghz, but only a single core was used for execution. The main calculations are 3 k-bit prime numbers, $p$, $q$ and $e$, and the inverse of $e$ as $d$.

| k-Bits | With Barrett Reduction | Without Barrett Reduction |
|---|---|---|
| 128 | 0.35s | 1.1s |
| 256 | 1.3s | 7.3s |
| 384 | 2.9s | 22s |
| 512 | 5.3s | 47s |
| 640 | 8.1s | 120s |
| 768 | 12s | 220s |
| 896 | 22s | 360s |
| 1024 | 32s | 540s |
| 1,152 | 45s | 900s |
| 1,280 | 61s | - Give up - |
| 1,408 | 76s | |
| 1,536 | 100s | |
| 1,664 | 130s | |
| 1,792 | 150s | |
| 1,920 | 230s | |
| 2,048 | 260s | |

Table 1: Key Generation Time (2 s.f.)

If we graph this data (see figure 3), we can see that the execution time rapidly increases past 1280 bit keys. In order to generate keys in a reasonable amount of time, we need further theoretical improvements to the algorithms.

### 4.1.1 Packing and Unpacking

Profiling revealed that pack, unpack and repack operations contribute less than 1.0% to total runtime.

### 4.1.2 Barrett Modular Reduction

Barrett modular reduction is used in the exponentiation function to reduce the amount of work done. Normal modulus function is performed using division. Barrett modular reduction allows us to perform a single division and use this to perform modulus function without division, thus improving performance significantly.

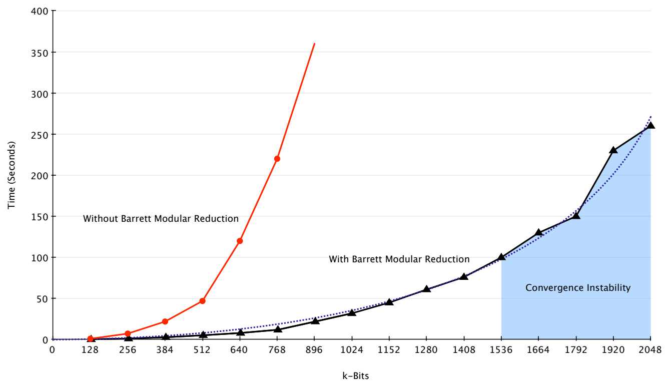

Figure 3: A graph of key generation performance, clearly showing exponential time complexity.

### 4.1.3 Convergence Instability

One observation made when testing the key generation performance was that as the key size increases not only does the time required increase, but also variance in the time required. While small keys quickly converged on an accurate measurement, large keys produced a wide variance in timing information.

As a result, the larger keys required repeated testing to get a useful result. In the future, it may prove useful to write an adaptive benchmark which continues to test key generation until the deviation in the average time taken is less than a specified tolerance; however this type of testing may require exponentially more tests for a reasonably accurate convergence for a problem such as key generation, making its use very costly.

## 4.2 Encryption and Decryption

Encryption and decryption is acceptable for small amounts of data and small key sizes. The actual performance is dictated by key size and packing method. If the key is bigger, $n$ is bigger, and we can pack more data into each message. However, numerical operations which have non-linear performance become increasingly expensive with larger keys.

| k-Bits | Bytes per Second |
|---|---:|
| 128 | 24KB/s |
| 256 | 17KB/s |
| 512 | 7.7KB/s |
| 1024 | 2.5KB/s |

Table 2: Speed of encryption and decryption.

Profiling the code revealed that the biggest improvement which could be made for large exponents is to improve the exponentiation function. As well as Barrett modular reduction, several options exist to improve algorithmic performance. Prior research suggests that using sliding window exponentiation could reduce the cost of exponentiation by reducing the number of steps required by caching low order exponents.

### 4.3 Division by Repeated Doubling

As an additional experiment, division was implemented by two different algorithms. Traditional method using fixed-precision division, and division by repeated doubling.

Division by repeated doubling is faster than traditional division algorithm, but only when the numerator and denominator are similar in magnitude. This is because in a few steps we can double the number and exceed the numerator. In the case where the numbers are further apart, the number of steps required increases more than the traditional implementation of division.

## 5 Conclusion

I have presented a moderately efficient implementation of RSA. However, there are many places where the algorithm can be improved. According to my testing, the next step is to improve speed of multiplication and also reduce the number of times it is performed during exponentiation. There are several ways to do this, such as Karatsuba multiplication and improved carry handling to improve algorithmic performance. Another option would be to use sliding window exponentiation which decreases the number of multiplications. It would also be possible to utilise more advanced vector operations and assembly language implementations of various operations to enhance execution speed.

As well as these major areas of performance enhancement, there are many minor areas that could be refined, however these have not been done due to time limitations and the fact that they are relatively minor part of the total run time of the program.