RSA Encryption in Java via BigInteger

Chris and David Etler, Ocean County College

Objectives

The objective of this prject was to implement the RSA encryption scheme in Java. We provide the following deliverables to exhibit knowledge of RSA and public key cryptograhpy:

1. Generation of 2048 RSA keys
2. Encryption of integers less than 2048 bits in length via RSA
3. Conversion of plaintext to integer representation (*padding scheme*)
4. A rudimentary shell for encryption of messages from files and reading of keys

Introduction

RSA cryptography is a industry-standard algorithm that implements public key cryptography in a simple yet extremely powerful way. Public key cryptography is a system which allows two parties to contact one another without and fear of interception, for if the message were to be intercepted it would mean nothing to to the interceptor. This sort of system is highly useful everywhere from espionage to online gaming, as it allows clients to communicate openly without fear of any signal interception and without ever needing to communicate a shared secret over a private channel.
High level, the system works thus:

1. Alice is trying to communicate with Bob secretly, while Eve is able to intercept any message that Alice sends to Bob or vice versa.
2. In order to communicate with Bob, Alice needs to chose a shared secret with Bob which must be used to both verify Alice's identity and encode a message such that Eve will have no means of deciphering it without knowledge of the shared secret.
3. Alice cannot communicate the shared secret openly with Bob, as there is no secure channel which Eve cannot overhear.
4. Bob publishes a item that can encrypt a message in such a way that only he will ever be able to unlock
5. This item is published freely such that both Alice and Eve have access to it. We can envision such a item as a padlock, which can be locked by anyone but only unlocked by one with the key.
6. Alice can now write her message to Bob and lock it with the padlock. Only Bob will be able to unlock the padlock despite everyone having access to it.
7. Alice also has her own padlock which can be used by Bob to send a message back to her.

There are a few technical considerations that must be taken into account to fully understand the significance of RSA. For one, we must develop a mathematical "padlock" in order to send messages abstractly rather than with physical padlocks. We will refer to these abstract padlocks as *public keys* and the keys used to unlock them as *private keys*, and both together as a *key pair*. Key pairs must be generated algorithmically and keys must be complex enough such that the lock can't be "picked" by Eve. The solutions to these technical hurdles as well as the inner workings of the RSA algorithm are presented in the section on the RSA algorithm.

Message Padding

In order to represent message plaintext as an integer for encryption, a *padding scheme* was necessary. Our padding scheme encodes an symbol set of 10 characters as numbers between 00-99. A message is simply a series of these number combinations terminating with a single three-digit number that encodes the length of the message, so that leading zeros aren't lost.

RSA Algorithm

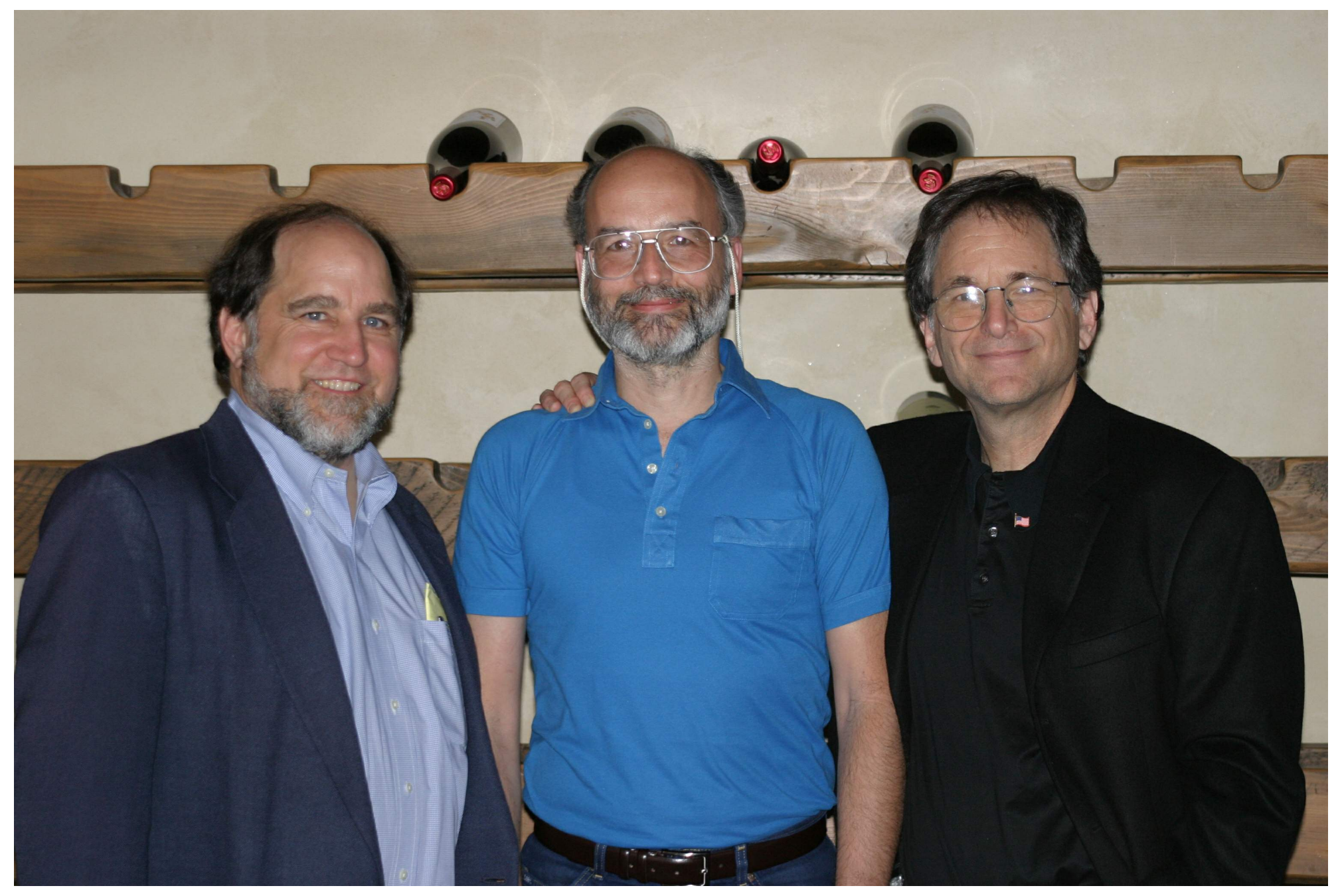
The RSA algorithm works by generating two keys: a public and a private key. In reality, these keys are simply very long prime numbers. The security of the algorithm depends completely on the length of these numbers. In the past, numbers as low as 768-bit have been used, but modern practices call for no lower than 1024-bit; some implementations use as high as 4096-bit keys. For this implementation we decided to use 2048-bit keys, which would mean prime numbers on the order of 2^{2048}.
The algorithm works as such:

1. First, two prime numbers p and q are chosen. These two numbers together are used to form the public key. Specifically, the public key is $pq = p * q$. Both p and q should be of the same bit-length for security, so for a 2048-bit key, we chose to use 1024-bit prime numbers.
2. A number e is chosen and agreed upon by all using the algorithm. This is called the public exponent and the standard is $e = 65537$, though in practice any number mutually prime with $(p-1)(q-1)$ is usable. 65537 is chosen as it's a prime number and is equal to $2^{2^4} + 1 = 1 \ll (1 \ll 4) + 1$, which can be easily calculated by a computer.
3. The private key d is found by taking the *modular inverse* of e mod $(p-1) * (q-1)$. In our program, this number if found via the *extended Euclidean algorithm*.
4. A message is chosen and represented as an integer called M.
5. From the message, a cypher-text C is generated. $C = M^e \ mod \ pq$.
6. The message is sent to the owner of the public key, who can decrypt the message using $M = C^d \ mod \ pq$.

The algorithm derives strength from a few different factors. First, it is very difficult to factor a large number into it's component primes. When a RSA public key is generated, two small prime numbers are generated separately and then multiplied by each-other to calculate the public key. It is easy for a computer to perform this task, taking mere seconds on even an antiquated laptop. However, such cannot be said about finding the two primes that make the public key given just the key itself. Even a modern supercomputer would be incapable of such a feat, even on the time-scale a years.
Further, RSA derives strength from how simple it is to calculate the modular inverse of $e \ mod \ pq$ given just $(p-1)(q-1)$. This process is done via the extended Euclidean algorithm. The algorithm will always calculate the inverse of `A` mod `B` so long as one exists. The output are numbers a, s, and t such that $a = GCD(A, B)$ and $A * s + B * t = a$. The second output, s, is the RSA decryption key when $A = e$ and $B = (p-1)(q-1)$. This can be calculated efficiently when p and q are originally known; however, finding this with only knowledge of the public key pq is a monumental task, making RSA extremely secure.

History

RSA was first described publicly in 1977 by a trio of MIT mathematicians named Ron **R**ivest, Adi **S**hamir, and Leonard **A**dleman, after whom the algorithm is named. Prior to that the algorithm was used internally by the UK intelligence agency GCHQ, where it was discovered independently by Clifford Cocks. Despite having been discovered 4 years earlier, the algorithm is attributed to the MIT trio for their work being the first published for the general public.



Ron Rivest, Adi Shamir, Len Adleman in 2003

Today, RSA is ubiquitous in many areas if security, mainly due to its simplicity and security. It is secure enough to be used by government espionage, yet easy enough to be implemented by amateurs.

RSA Algorithm Code

```
public final class RSA {

    static final Message textConverter = new Message();

    public static BigInteger encrypt(String M, BigInteger e, BigInteger pq) {
        BigInteger Mn = textConverter.toInt(M);
        return encrypt(Mn, e, pq);
    }
    public static BigInteger decrypt(String C, BigInteger d, BigInteger pq) {
        BigInteger Cn = textConverter.toInt(C);
        return encrypt(Cn, d, pq);
    }

    public static BigInteger[] generateKey() {
        Random rand = new Random();
        BigInteger p = BigInteger.probablePrime(2048, rand);
        BigInteger q = BigInteger.probablePrime(2048, rand);
        BigInteger[] key = {p.multiply(q), decryptionKey(p, q)};
        return key;
    }

    public static BigInteger decryptionKey(BigInteger p, BigInteger q) {
        BigInteger v = p.subtract(BigInteger.valueOf(1)).multiply(q.subtract(BigInteger.valueOf(1)));
        BigInteger d = Euclidean.extendedEuclidean(BigInteger.valueOf((1<<(1 << 4)) + 1), v)[1];
        if (d.compareTo(BigInteger.valueOf(0)) < 0) {
            d = d.add(v);
        }
        return d;
    }
}

public class Euclidean {
    public long[] extendedEuclidean(long A, long B) {
        long a = A, b = B, s = 1, t = 0;
        long u = 0, v = 1, r = 0, q = 0, nu, nv;
        while (b != 0) {
            r = a % b;
            q = (long) Math.floor(a / b);
            a = b;
            b = r;
            nu = s - u * q;
            nv = t - v * q;
            s = u;
            t = v;
            u = nu;
            v = nv;
        }
        // gcd == a
        long[] m = {a, s, t};
        return m;
    }
}
```

Conclusion

This project served as great learning experience. Understanding and implementing the RSA algorithm is simple enough to be accessible to a student at a community college, yet ubiquitous and powerful enough to make a valuable chuck of knowledge. The implementation here is not complete, however. In practice a better padding scheme which adds entropy to the message is necessary to thwart attempts to decipher messages by accumulating a codex. Further, messages by signing by a second set of keys is common in practice to ensure that all messages were sent by the proper party. Any legitimate software application would be better off utilizing well-used RSA libraries which have been tested by many. However, for a hobby project, this program has been very useful to help understand the inner workings of the RSA algorithm.

Acknowledgments

We would like to thank Prof. Pezzimenti for his support in this project. We would also like to thanks as our parents for providing room and board pro bono while we conduct our studies. Further we would like to thanks Linus Torvalds for writing Git and the creators of NetBeans because they're seriously useful tools for development of applications in Java. Finally we'd like to thank Donald Knuth for writing LaTeXand Brian Amberg for creating `baposter`, the library through which this poster was rendered.