A Survey of Cryptographic Algorithms
Shelley Kandola
May 13, 2013
Advisor: Dr. Lisa Torrey

Submitted in partial fulfillment of the requirements for honors in computer science, Department of Mathematics, Computer Science, and Statistics, St. Lawrence University.
Abstract

In this paper, we survey a variety of cryptographic algorithms falling into four main categories: DES (block cipher), RC4 (stream cipher), SHA-1 (hash), and RSA (public-key cryptography). We provide a Python implementation for each of these algorithms. Then we go into further detail on RSA cryptography. We analyzed the algorithmic complexity of several procedures for generating RSA keys, chose the most efficient of each, and did implementations in Python and in Java. We compared the implementations by measuring how long they took to generate keys up to 500 digits long.
## Contents

Abstract 2

1 Cryptographic Algorithms 5
  1.1 Symmetric Algorithms 6
  1.2 Public Key Cryptography 7
  1.3 Hashes 7

2 Implementations 8
  2.1 DES 8
  2.2 RC4 9
  2.3 SHA-1 10
  2.4 RSA 12

3 An Analysis of RSA 14
  3.1 Algorithms 14
     3.1.1 Primality Testing 15
     3.1.2 Prime Generation 15
     3.1.3 Modular Exponentiation 16
     3.1.4 Multiplicative Inverses 19
  3.2 Data 19
  3.3 Analysis 20
  3.4 Conclusion 22

Appendices 23

A Code 23
  A.1 DES 23
  A.2 RC4 33
  A.3 SHA-1 36
  A.4 RSA - Python 40
     A.4.1 Prime Generation 40
A.4.2 RSA Helper .................................................. 45
A.4.3 RSA .......................................................... 47
A.5 RSA - Java ....................................................... 48
A.5.1 Key Generation .............................................. 48
A.5.2 RSA Math ..................................................... 55
A.5.3 RSA ........................................................... 57

Bibliography .......................................................... 59
1 Cryptographic Algorithms

Cryptographic algorithms are a means of discreetly conveying secret information between one party and another. In general, a plaintext message is encrypted using a cryptographic algorithm. Through encryption, the original message becomes ciphertext and its original content is completely concealed. The ciphertext can then be sent safely to the recipient. When the recipient is ready to reveal the message, he or she can do so by applying a decryption algorithm, which will reveal the original plaintext. Only the recipient can apply the decryption algorithm because, ideally, only the recipient knows the keys necessary for decrypting the ciphertext. Keys are used to personalize and secure a cryptographic algorithm to only the sender and recipient.

Figure 1: Concept of cryptography

Secure encryption requires two properties: confusion and diffusion. Confusion refers to complicating the relationship between the ciphertext and the key as much as possible,
whereas **diffusion** refers to complicating the relationship between plaintext and ciphertext as much as possible (See [4]). One way to achieve confusion is through substitution: the replacement of characters with other characters. The Caesar cipher, for example, does a simple shift substitution. If the Caesar cipher is set for A=S, B=T, C=U, etc., then we can perform the following encryption:

\[
\text{CAT} \rightarrow \text{USL}
\]

One way to achieve diffusion is by jumbling the symbols of a plaintext message. For example, a transposition cipher is done by writing the letters vertically side-by-side in columns of a fixed height, and then concatenating the rows from top to bottom to form the new message. The message “The crow flies at dawn” can be written in columns of height 4 the following way:

\[
\begin{array}{cccc}
T & R & L & A & W \\
H & O & I & T & N \\
E & W & E & D \\
C & F & S & A \\
\end{array}
\Rightarrow \text{TRLAWITNEWECDFA}
\]

Most cryptographic algorithms involve a combination of both confusion and diffusion.

### 1.1 Symmetric Algorithms

As far as encryption and decryption are concerned, there are two types of key-based algorithms. **Symmetric algorithms** are key-based algorithms that use the same key for encryption and decryption, or whose encryption and decryption keys can be generated from each other. Symmetric key algorithms therefore require that the key be kept a secret. The Caesar cipher mentioned in the previous section is a very simple example. Both block ciphers and stream ciphers fall into this category. To an extent, hash algorithms fall into this category, although there is no decryption function for hash algorithms, so we will discuss that later.

The difference between block ciphers and stream ciphers lies in the size of the message chunks on which the cipher operates. **Stream ciphers** are symmetric algorithms that operate on plaintext input one bit (or byte) at a time. **Block ciphers** are symmetric algorithms that operate on larger blocks, usually 64 bits in size. Later in this paper we will discuss the specifics of two such algorithms, DES and RC4.
1.2 Public Key Cryptography

Public key algorithms differ from symmetric algorithms in that different keys are used for encryption and decryption. Whereas keys used in symmetric algorithms can be generated from each other, keys used in public key algorithms cannot; the sender and the recipient of the message each have their own private key that works with the public key shared between them. Further on in this paper we discuss a specific public key algorithm, RSA, in depth.

1.3 Hashes

Hashes do not quite fall into either of the above categories because they are one-way functions; there is no decryption algorithm. Hash functions map variable-length input to fixed-length output. A common operation used in hash functions is the XOR (⊕) operator. For example, if \( a \oplus b = 1 \), it is impossible to determine which bits of \( a \) and \( b \) are valued 0 and which are valued 1 (see table 1). This explains in part why a message cannot be revealed given a hashed message.

\[
\begin{array}{ccc}
X & Y & X \oplus Y \\
1 & 1 & 0 \\
1 & 0 & 1 \\
0 & 1 & 1 \\
0 & 0 & 0 \\
\end{array}
\]

Table 1: An XOR truth table.

Hash algorithms are often used for verification and validation rather than message passing. A checksum is the hashed value of a block of data - for example, the source code for a piece of software. If a user wants to determine if their copy of software is genuine, they can check its checksum against the checksum provided by software company. Many website databases also store hashed passwords instead of the passwords themselves. This prevents a hacked database from revealing passwords. An important problem for hash algorithms is the risk of collisions, in which two messages have the same hashed output. If, for example, two distinct confidential messages hashed to the same value, a court of law would not be able to reliably prove which document was the “original.”
2 Implementations

In this section, we describe specific examples of the four types of cryptographic algorithms: DES (a block cipher), RC4 (a stream cipher), SHA-1 (a hash algorithm) and RSA (a public-key algorithm). Python code for each of these algorithms is included in the appendices.

2.1 DES

The Data Encryption Standard is an old (1972) symmetric block cipher that was commissioned by the NSA to serve as a standard for data encryption. The design of DES required a handful of features, the most interesting of which is the fact the security of the algorithm lies in the key itself and not in the secrecy of the algorithm. That is, even if a hacker knows the entire inner-workings of the algorithm, they should not be able to decipher a message unless they know the key. More standard requirements include that the algorithm be easy to implement for both hardware and software[2].

As a block cipher, DES operates on chunks of plaintext 64 bits at a time. It requires a 56-bit key and at the simplest level, applies sixteen rounds of confusion and diffusion, using 16 keys generated from the original 56-bit key. The algorithm at the outermost level is shown in Figure 2.1. Each round within the algorithm performs the following steps:

1. Expansion Permutation: 32 bit chunks of the message are expanded to 48 bits
2. XOR with round key: the round-specific key is XORed with the message
3. S-Box Substitution: each byte of the message is replaced with another byte
4. P-Box Permutation: the 48-bit message chunk is scrambled and reduced to 32 bits

The permutations are types of diffusion and the substitution is a type of confusion; they are performed using fixed arrays of numbers called P-Boxes and S-Boxes. An S-Box refers to a substitution box, which works by substituting a byte $b$ of the message with the $b^{th}$ index of the S-Box. A P-Box permutation works by replacing byte $b$ of a message with whatever byte is at the index specified by P-Box$[b]$. These steps are individually simple, but there are eight different S-Boxes used throughout the algorithm, and the left and right halves of the message are swapped between iterations of the rounds. Despite
all the security measures, DES has been cracked and is no longer in use. Because the key
size for DES is so small, brute force attacks have proven effective at cracking DES.

2.2 RC4

The symmetric algorithm RC4 is a simple stream cipher developed by Ron Rivest in
1987[2]. As a stream cipher, it operates on the plaintext message one byte at a time. It
relies solely on confusion and requires an $8 \times 8$ S-Box that is generated using algorithm
1, as well as a 256-byte array that stores the bytes of a key repeated as many times as
needed to fill the entire array $K_0, K_1, \ldots, K_{255}$. Using the S-Box and the key array, a
pseudorandom sequence of numbers is generated that is then XORed with the plaintext
Because the key generates the S-Box that supplies the pseudorandom bytes, a message encrypted with RC4 can only be decrypted with the key-generated S-Box as well. RC4 is a very simple algorithm and is therefore not used for highly classified data. It is used in programs such as Oracle Secure SQL and is part of the Cellular Digital Packet Data specification.[2]

### 2.3 SHA-1

SHA-1 stands for “Secure Hash Algorithm 1,” and was first published for use in 1995. One of its most amazing feats is taking a message of variable-length input (up to $2^{64}$ bits long), and reducing to to 160-bit encrypted output[3]. At a glance, SHA-1 requires some preparation of the plaintext message followed by 80 rounds of encryption. Within each round (shown in Figure 4), the functions and constants used change depending on the round number.

In algorithm 2, the original message (of length $< 2^{64}$ bits) is divided into blocks of bit-length 512. The 160-bit digests that come out of each iteration of the algorithm for each block are then XORed together at the very end. SHA-1 cannot be reversed for a variety of reasons. Input of any length maps to an output of fixed length (160 bits). Within the algorithm, there are more operations that have multiple input combinations yielding the same output, for example, XOR. This leads to the possibility of collision attacks, in which
Algorithm 1 RC4 Encryption and Decryption

Require: Key array \( K \)

1: procedure GenSBox \( \triangleright \) Populates the S-Box based on the key
2: \( S = \) empty list
3: for \( i \) from 0 to 255 do
4: \( j \leftarrow (j + S_i + K_i) \mod 256 \)
5: \( S_i, S_j \leftarrow S_j, S_i \) \( \triangleright \) Swap the bytes
6: end for
7: return \( S \)
8: end procedure

Require: S-Box

9: procedure RandByte \( \triangleright \) Generates a byte \( K \) to \( \oplus \) with message
10: \( i \leftarrow (i + 1) \mod 256 \)
11: \( j \leftarrow (j + S_i) \mod 256 \)
12: \( S_i, S_j \leftarrow S_j, S_i \) \( \triangleright \) Swap the bytes
13: \( t \leftarrow (S_i + S_j) \mod 256 \)
14: \( K \leftarrow S_t \)
15: return \( K \)
16: end procedure

Figure 4: A round within SHA-1
Algorithm 2 SHA-1

1: procedure SHA-1(M) \(\triangleright\) Pass in the array of 512 bits containing the message
2: \[ W_0 \ldots W_{16} \leftarrow \text{the message divided into sixteenths} \]
3: for \( t \) from 16 to 70 do
4: \[ (W_t \leftarrow W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-15}) \ll 1 \]
5: end for
6: \( a, b, c, d, e \) \(\triangleright\) \(a, b, c, d, e\) are preset constants
7: for \( t \) from 0 to 79 do
8: \( \triangleright f, K \) are functions and constants that vary based on \( t \)
9: \[ \text{tmp} \leftarrow a \ll 5 + f_t(b, c, d) + e + W_t + K_t \]
10: \( e, d \leftarrow d, c \)
11: \( c \leftarrow b \ll 30 \)
12: \( b, a \leftarrow a, \text{tmp} \)
13: end for
14: return \( a, b, c, d, e \)
15: end procedure

\[ \text{different plaintext messages hash to the same digest.} \]

In 2005, SHA-1 was found to be weak against collision attacks. Although no certain collisions have been found, collisions have been found on smaller versions of the problem, e.g., in implementation that use fewer rounds. The possibility of collision attacks was enough to phase out the use of SHA-1. Since then, SHA-224, SHA-256, SHA-384, and SHA-512 have been published and implemented for general use.[3] SHA-1 is still used often, however, on data that does not require a security clearance.

2.4 RSA

The RSA encryption/decryption algorithms are a type of public key cryptography named for Ron Rivest, Adi Shamir and Leonard Adleman, who first presented the algorithm in 1977. Encryption and decryption are both very simple: If Alice is sending a secret message \( m \) to Bob, he tells her his public key, which consists of two values, \( e \) and \( n \). Alice then sends to Bob the ciphertext message \( c = m^e \mod n \). He decodes it using his private key, \( d \), computing \( m = c^d \mod n \) to get Alice’s original message back. The eavesdropper Eve cannot decode it because she does not have Bob’s private key. However, she could figure out \( d \) if she could factor \( n \), so the algorithm’s security relies on generating very large prime numbers that are difficult to factor.

We now show why Bob’s decryption procedure reveals the message \( m \). If \( c = m^e \mod n \)
Algorithm 3 RSA Key Generation

1: procedure GETKEYS(b) \(\triangleright\) Generates keys based on \(b\)-bit primes
2: \(p, q \leftarrow\) primes of bit-length \(b\)
3: \(n \leftarrow p \cdot q\) \(\triangleright\) First public key
4: \(\phi \leftarrow (p-1)(q-1)\)
5: \(e \leftarrow e\) such that \(\text{gcd}(e, \phi) = 1\) \(\triangleright\) Second public key
6: \(d \leftarrow e^{-1} \mod \phi\) \(\triangleright\) Private key
7: return \(n, e, d\)
8: end procedure
and Bob computes $c^d \mod n$, then he gets:

$$m^{e^d} \mod n = m^{e \cdot d} \mod n = m^{ \mod \phi(n) \mod n} = m.$$  \hspace{1em} (1)

To understand the last few steps of equation 1, we must first understand a function called Euler’s $\phi$-function.

**Definition.** Euler’s $\phi$-function, called $\phi(n)$, returns the number of integers less than or equal to $n$ that are relatively prime to $n$. If $n$ has a prime factorization $n = p_1^{e_1} \cdots p_k^{e_k}$ then $\phi(n) = \prod_{i=1}^{k} (p_i - 1)(p_i^{e_i} - 1)$.

There exists a property of modular exponentiation that states if $\gcd(a, m) = 1$, then $a^{\mod \phi(m)} \mod m \equiv a$. Therefore, equation 1 will hold when the keys $n$, $e$, and $d$ are computed as shown in Algorithm 3.

### 3 An Analysis of RSA

After surveying the previous four types of cryptographic algorithms, we decided to focus on one algorithm in particular: RSA. We began by analyzing its algorithmic complexity, which we express using Big-O notation. **Big-O notation** is used to classify algorithm runtimes by their dominant parts. Coefficients and polynomial terms of lesser degree are often dropped when using Big-O notation. For example, if an algorithm executes a loop of length $n$ three times, instead of describing it as having a runtime of length $3n$, we would say it runs in $O(n)$ time, or linear time. If an algorithm can be described as having a runtime of $n^3 + 2n^2$, we would say it runs in $O(n^3)$ or cubic time. The two main groups of algorithm runtimes are polynomial and exponential. Exponential-time algorithms are considered impractical, since they run extremely slowly on large inputs, while polynomial-time algorithms scale better.

#### 3.1 Algorithms

Performing RSA encryption and decryption requires two main techniques: key generation and modular exponentiation. The key generation is costly because it entails the generation of two large primes, which must be done efficiently. Depending on the desired security of the implementation, the primes are commonly 1024 or 2048 bits in length. Such large
primes are needed in order to make factoring the public key $n$ near impossible, and therefore the discovery of the private key. Modular exponentiation may seem simple, but when dealing with numbers as large as those used in RSA, efficiency also becomes a concern.

3.1.1 Primality Testing

Primality Testing is an important feature in prime number generation. The goal of a primality checker is to determine if a given number is prime. In order to do this efficiently, however, sometimes accuracy needs to be traded off for speed.

Algorithm 4 gives the naive procedure for determining if a number is prime. This algorithm has a 0% error rate, but runs in $O(2^n)$ time, where $n$ is the bit-length of $p$. Although this algorithm will return accurate results, it has an exponential runtime. Since RSA requires primes between 1024 and 2048 bits, checking the primality of these numbers would be impossibly costly.

Algorithm 4 Basic Primality Test

1: procedure isPrime($p$)
2: for $i \in [2, \sqrt{p}]$ do
3: if $i$ divides $p$ then
4: return false
5: end if
6: end for
7: return true
8: end procedure

Instead, we can opt for a primality checker with a small possibility of error, but a much faster runtime. Algorithm 5 is built off of the convenient fact that if $p$ is prime, we have that for all $a < p$, $a^{p-1} \mod p = 1$. This is known as Fermat’s little theorem.

Algorithm 5 only runs in $O(n^2)$ time, where $n$ is the bit-length of $p$. While this algorithm is much faster than algorithm 4, it has an error probability of $\frac{1}{2^k}$, where $k$ is the amount of random $a$ values checked. The probability of error becomes negligible for even modest values of $k$, such as 100.

3.1.2 Prime Generation

There are a few general approaches to generating a prime number. If the prime-length we are looking for is not too large, we can use the Sieve of Atkin to generate all primes
Algorithm 5 Fermat’s Primality Test

1: procedure isPrime(p)
2: for k times do
3: \( a \leftarrow \text{random integer such that } 1 \leq a < p \)
4: if \( a^{p-1} \equiv 1 \mod p \) then
5: return true
6: end if
7: end for
8: return false
9: end procedure

in the desired range, shown in algorithm 7. For larger primes, we could choose randomly from all numbers of appropriate length until we find one that passes the primality test, or we could generate a list of likely prime numbers and then test candidates from that. Both approaches are reasonable, but the latter is slightly faster.

Algorithm 6 Simple Prime Generation

1: procedure PrimeGen(b) \(\triangleright\) Finds a prime with bit-length b
2: \( p \leftarrow \text{random number of bit length } b \)
3: while \( \neg \text{isPrime}(p) \) do
4: \( p \leftarrow \text{random number of bit length } b \)
5: end while
6: return \( p \)
7: end procedure

Algorithm 6 gives the most basic method for generating a prime number. Its runtime is dependent on that of the isPrime() function called within it and on the density of the primes. As a larger prime is required, checking the primality takes longer and the prime density of numbers of that length decreases.

We created a combination of the algorithms 6 and 7 to generate the primes used in our RSA implementations. This is presented in algorithm 8. This algorithm uses the sieve to generate all small primes, and then deletes multiples of these small primes from a predetermined range of candidate primes that have the appropriate digit-length. Finally, it applies the primality test to the remaining candidates until it finds one that passes.

3.1.3 Modular Exponentiation

Modular exponentiation is used frequently throughout the RSA algorithm. Primality checking, prime generation, and encryption and decryption all used modular exponen-
Algorithm 7 Sieve of Atkin

1: procedure SIEVE(limit) \Comment{Generates primes ≤ limit}
2:     isPrime(i) ← false, ∀i ∈ [5, limit] \Comment{initialize the sieve}
3:     \Comment{Candidate primes take a certain quadratic form}
4:     for (x, y) ∈ [1, \sqrt{\text{limit}}] \times [1, \sqrt{\text{limit}}] do
5:         n ← 4x^2 + y^2
6:         if (n ≤ limit) and (n mod 12 ≡ 1 or n mod 12 ≡ 5) then
7:             isPrime(n) ← ¬isPrime(n)
8:         end if
9:         if (n ≤ limit) and (n mod 12 ≡ 7) then
10:            isPrime(n) ← ¬isPrime(n)
11:        end if
12:        if (x > y) and (n ≤ limit) and (n mod 12 ≡ 11) then
13:            isPrime(n) ← ¬isPrime(n)
14:        end if
15:     end for
16:     \Comment{“Sieve” out composites}
17:     for n ∈ [5, \sqrt{\text{limit}}] do
18:         if isPrime(n) then
19:             for k ∈ \{n^2, 2n^2, 3n^2, \ldots\} do
20:                 isPrime(n) ← false
21:             end for
22:         end if
23:     end for
24:     return isPrime
25: end procedure
Algorithm 8 Efficient Prime Generation

Require: smallPrimes \(\triangleright\) All primes less than 65000 generated by Sieve of Atkin

1: procedure PrimeGen\( (n) \) \(\triangleright\) Generate a prime \(n\) digits long
2: \hspace{1em} lower \(\leftarrow 10^{n-1}\)
3: \hspace{1em} upper \(\leftarrow 10^n - 1\)
4: \hspace{1em} \(r \leftarrow \text{random} \in \{ \text{lower}, \text{upper} \} \)
5: \hspace{1em} primes \(\leftarrow\) empty map \(\triangleright\) For storing all candidate primes
6: \hspace{1em} if \(r\) is even then
7: \hspace{2em} \(r \leftarrow r - 1\)
8: \hspace{1em} end if
9: \hspace{1em} for \(k\) times do \(\triangleright\) \(k = 10000\) is common
10: \hspace{2em} primes\([r]\) \(\leftarrow true\)
11: \hspace{1em} \(r \leftarrow r + 2\)
12: \hspace{1em} end for
13: \hspace{1em} for \(j \in\) smallPrimes do
14: \hspace{2em} for all multiples of \(j < r + k\) do
15: \hspace{3em} primes\([\text{multiple}]\) \(\leftarrow false\)
16: \hspace{2em} end for
17: \hspace{1em} end for
18: return primes
19: end procedure

Algorithm 9 Simple Modular Exponentiation

1: procedure modPow\( (x, y, m) \)
2: \(z \leftarrow 1\)
3: \(\text{for}~ y\) times do
4: \hspace{1em} \(z \leftarrow x \cdot z \mod m\)
5: \hspace{1em} end for
6: end procedure

Algorithm 9 is the naive method for performing modular exponentiation. While simple to implement, it runs in \(O(2^n)\) time, where \(n\) is the bit-length of \(y\). Modular exponentiation can also be computed more efficiently, as shown in algorithm 10. This algorithm only runs in \(O(n^3)\) time, where \(n\) is the bit-length of \(y\). A problem with this implementation, however, is that recursion depth can be exceeded for very large values of \(y\). Consequently, algorithm 10 must be implemented iteratively, but with the same logic.
Algorithm 10 Fast Modular Exponentiation

1: procedure modPow(x, y, m)
2:     if y = 0 then
3:         return 1
4:     end if
5:     z ← modPow(x, \(\frac{y}{2}\), m)
6:     if y is even then
7:         return \(z^2 \mod m\)
8:     else
9:         return \(x \cdot z^2 \mod m\)
10:    end if
11: end procedure

3.1.4 Multiplicative Inverses

Only one subroutine of RSA needs to compute multiplicative inverses, but as the desired prime length increases, this step could be costly. Recall in algorithm 3 that \(d\) is the multiplicative inverse of \(e \mod \phi\). As with most algorithms, there are simple, inefficient and complicated, efficient ways to compute multiplicative inverses.

Algorithm 11 Simple multiplicative inverse computation

1: procedure INVERSE(a, m) ▶ Compute \(a^{-1} \mod m\)
2:     for all \(1 \leq b < m\) do
3:         if \(a \cdot b \equiv 1 \mod m\) then ▶ Brute-force check all possibilities
4:             return \(b\)
5:         end if
6:     end for
7: end procedure

Algorithm 11 is clearly very straightforward and easy to implement, but has a runtime of at least \(O(2^n)\) where \(n\) is the bit-length of \(m\). As \(m\) increases, the time to search for a multiplicative inverse increases exponentially. Also, if \(\gcd(a, m) \neq 1\), it is certain that no multiplicative inverse will ever be found. To arrive at this realization sooner, a faster algorithm can be used, shown in Algorithm 12.

3.2 Data

After determining efficient algorithms for performing all the functions required by RSA, we coded them in both Python and Java. Table 2 gives the runtime, in seconds, of the section of the Python implementation that generates two \(n\)-digit primes. Similarly, table
Algorithm 12 Euclid’s Extended Algorithm

1: procedure extendedGCD(a, b)
2: \( x, lastx \leftarrow 0, 1 \)
3: \( y, lasty \leftarrow 1, 0 \)
4: while \( b \neq 0 \) do
5: \( q \leftarrow \left\lfloor \frac{a}{b} \right\rfloor \)
6: \( a, b \leftarrow b, a \mod b \)
7: \( x, lastx \leftarrow lastx - q \cdot x, x \)
8: \( y, lasty \leftarrow lasty - q \cdot y, y \)
9: end while
10: return lastx
11: end procedure

The multiplicative inverse needs to be a positive number

12: procedure inverse(a, m)
13: \( inv \leftarrow \text{extendedGCD}(a, m) \)
14: if \( inv < 0 \) then
15: return \( inv + m \)
16: else
17: return \( inv \)
18: end if
19: end procedure

3 gives the runtime, in seconds, of the section of the Java implementation that generates two \( n \)-digit primes.

The data displayed here was collected on a 32-bit Windows 7 Enterprise operating system with an Intel Core 2 Duo processor running at 2.67 GHz and with 2.00 GB of RAM.

3.3 Analysis

Table 4 shows the average runtimes for both implementations. Some trial-and-error analysis shows that these runtimes are well fit by the line \( time = n^{2.5} \cdot 10^{-5} \). Falling in between \( O(n^2) \) and \( O(n^3) \), this leaves our prime generation with polynomial runtime, approximately \( O(n^{2.5}) \). As subroutines of our implementation run in both \( O(n^2) \) and \( O(n^3) \), this runtime approximation is as expected.

RSA requires primes between 1024 and 2048 bits long. This translates roughly to primes between 309 and 617 digits long. Our implementations generated 500-digit primes in just over 60 seconds, which is not an unreasonable time to wait, seeing as the primes (and therefore the keys) only need to be generated once. Once someone has his or her
Table 2: Runtime of Python implementation for generating two $n$-digit primes, in seconds

<table>
<thead>
<tr>
<th>$n$</th>
<th>Python 1</th>
<th>Python 2</th>
<th>Python 3</th>
<th>Python 4</th>
<th>Python 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.128</td>
<td>0.121</td>
<td>0.138</td>
<td>0.122</td>
<td>0.125</td>
</tr>
<tr>
<td>10</td>
<td>0.124</td>
<td>0.123</td>
<td>0.120</td>
<td>0.127</td>
<td>0.123</td>
</tr>
<tr>
<td>20</td>
<td>0.135</td>
<td>0.129</td>
<td>0.141</td>
<td>0.133</td>
<td>0.128</td>
</tr>
<tr>
<td>50</td>
<td>0.264</td>
<td>0.283</td>
<td>0.243</td>
<td>0.259</td>
<td>0.288</td>
</tr>
<tr>
<td>100</td>
<td>0.916</td>
<td>0.934</td>
<td>0.911</td>
<td>0.838</td>
<td>0.735</td>
</tr>
<tr>
<td>150</td>
<td>2.273</td>
<td>1.936</td>
<td>2.466</td>
<td>2.155</td>
<td>2.139</td>
</tr>
<tr>
<td>200</td>
<td>4.128</td>
<td>4.065</td>
<td>4.698</td>
<td>3.842</td>
<td>4.716</td>
</tr>
<tr>
<td>300</td>
<td>14.593</td>
<td>13.489</td>
<td>15.979</td>
<td>12.666</td>
<td>16.970</td>
</tr>
<tr>
<td>400</td>
<td>36.854</td>
<td>35.143</td>
<td>30.417</td>
<td>46.126</td>
<td>29.017</td>
</tr>
<tr>
<td>500</td>
<td>59.653</td>
<td>74.380</td>
<td>60.140</td>
<td>72.417</td>
<td>73.481</td>
</tr>
</tbody>
</table>

Table 3: Runtime of Java implementation for generating two $n$-digit primes, in seconds

<table>
<thead>
<tr>
<th>$n$</th>
<th>Java 1</th>
<th>Java 2</th>
<th>Java 3</th>
<th>Java 4</th>
<th>Java 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.079</td>
<td>0.041</td>
<td>0.063</td>
<td>0.057</td>
<td>0.036</td>
</tr>
<tr>
<td>10</td>
<td>0.1</td>
<td>0.047</td>
<td>0.042</td>
<td>0.035</td>
<td>0.032</td>
</tr>
<tr>
<td>20</td>
<td>0.102</td>
<td>0.061</td>
<td>0.05</td>
<td>0.049</td>
<td>0.055</td>
</tr>
<tr>
<td>50</td>
<td>0.205</td>
<td>0.162</td>
<td>0.148</td>
<td>0.153</td>
<td>0.16</td>
</tr>
<tr>
<td>100</td>
<td>0.805</td>
<td>0.605</td>
<td>0.587</td>
<td>0.614</td>
<td>0.619</td>
</tr>
<tr>
<td>150</td>
<td>1.693</td>
<td>1.909</td>
<td>1.744</td>
<td>1.656</td>
<td>1.677</td>
</tr>
<tr>
<td>200</td>
<td>3.831</td>
<td>3.817</td>
<td>3.796</td>
<td>3.330</td>
<td>5.244</td>
</tr>
<tr>
<td>300</td>
<td>13.282</td>
<td>12.366</td>
<td>17.784</td>
<td>12.019</td>
<td>12.944</td>
</tr>
<tr>
<td>500</td>
<td>70.324</td>
<td>71.385</td>
<td>58.751</td>
<td>52.962</td>
<td>67.164</td>
</tr>
</tbody>
</table>

Table 4: Comparison of average runtimes of Python and Java implementations

<table>
<thead>
<tr>
<th>$n$</th>
<th>Python</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.126</td>
<td>0.055</td>
</tr>
<tr>
<td>10</td>
<td>0.123</td>
<td>0.051</td>
</tr>
<tr>
<td>20</td>
<td>0.133</td>
<td>0.063</td>
</tr>
<tr>
<td>50</td>
<td>0.267</td>
<td>0.165</td>
</tr>
<tr>
<td>100</td>
<td>0.866</td>
<td>0.646</td>
</tr>
<tr>
<td>150</td>
<td>2.194</td>
<td>1.735</td>
</tr>
<tr>
<td>200</td>
<td>4.289</td>
<td>4.004</td>
</tr>
<tr>
<td>300</td>
<td>14.739</td>
<td>13.679</td>
</tr>
<tr>
<td>400</td>
<td>35.511</td>
<td>32.048</td>
</tr>
<tr>
<td>500</td>
<td>68.014</td>
<td>64.117</td>
</tr>
</tbody>
</table>
public and private keys, encrypting and decrypting the messages runs relatively quickly; it will run as fast as the fastest modular exponentiation implementation.

### 3.4 Conclusion

The purpose of this research was to learn about cryptographic algorithms and to analyze the efficiency of the RSA encryption algorithm when implemented with different subroutines and in different languages. Different subroutines trade off simplicity for efficiency. After determining efficient subroutines, we implemented RSA in both Python and in Java. We found that the two implementations had very similar run times. We found this surprising since we had expected the Java implementation to be significantly faster than the Python implementation. Python is an interpreted language, but this disadvantage may have been offset by a few fast behind-the-scenes tricks such as dynamic type conversions between small and large integers and fast built-in arithmetic for large integers. Future work could include implementing the algorithm in languages that better allow bit-level manipulation, such as C.
Appendices

A Code

A.1 DES

# DES.py
# Shelley Kandola
# Crypto SYE

def DES(message, k):
    global left
    global right
    global key
    key = k
    # Start with 64 bits
    # Apply Initial Permutation
    enc = permute(message, init_perm)
    # Split into two 32-bit chunks
    left, right = split(enc)
    # Send each of those chunks through the round 16 times
    schedule_keys(k)
    for i in range(16):
        des_round(left,right,key,i)
    # Merge the two 32-bit halves
    enc = merge(left,right)
    # Apply inverse of initial permutation
    enc = permute(enc, init_perm_inv)
    return enc

def DES_inv(message, k):
    global left
    global right

23
global key
ek = k
# Start with 64 bits
# Apply Initial Permutation
en = permute(message, init_perm)
# Split into two 32-bit chunks
left, right = split(enc)
# Send each of those chunks through the round 16 times
for i in range(16):
    des_round_inv(left, right, key, 15 - i)
# Merge the two 32-bit halves
en = merge(left, right)
# Apply inverse of initial permutation
en = permute(en, init_perm_inv)
return en

def schedule_keys(k):
global key
ek = k

###### MAKING THE KEY ############
for r in range(16):
    left_key, right_key = split(key)
    l_key_rot = rotate(left_key, key_shift[r])
    r_key_rot = rotate(right_key, key_shift[r])
    round_key = merge(l_key_rot, r_key_rot)
    round_key = permute(round_key, key_comp)
    key_schedule.append(round_key)
    # Set global key for next round
    global key
    key = merge(l_key_rot, r_key_rot)

24
def des_round(lh, rh, k, r):
    # Make copy of right half
    old_right = copy(rh)
    # Expansion Permutation on right
    new_right = permute(rh, exp_perm)
    # XOR half2 with key
    new_right = XOR(new_right, key_schedule[r])
    # Apply SBoxes to half2
    new_right = s_box_compression(new_right)
    # Apply PBox permutation
    new_right = permute(new_right, p_box)
    # XOR half2 with half1
    new_right = XOR(new_right, lh)
    print bin_to_hex(old_right), bin_to_hex(new_right), bin_to_hex(key_schedule[r])
    # Set Final Values
    if r<15:
        global left
        left = old_right
        global right
        right = new_right
    else:
global left
left = new_right
global right
right = old_right

def des_round_inv(lh, rh, k, r):
    # Make copy of right half
    old_right = copy(rh)

    # Expansion Permutation on right
    new_right = permute(rh, exp_perm)

    # XOR half2 with key
    new_right = XOR(new_right, key_schedule[r])

    # Apply SBoxes to half2
    new_right = s_box_compression(new_right)

    # Apply PBox permutation
    new_right = permute(new_right, p_box)

    # XOR half2 with half1
    new_right = XOR(new_right, lh)
    print bin_to_hex(old_right), bin_to_hex(new_right), bin_to_hex(key_schedule[r])

    # Set Final Values
    if r>0:
        global left
        left = old_right
        global right
right = new_right
else:
    global left
    left = new_right
    global right
    right = old_right

# Rotates a message 'm' left by 'i' bits
def rotate(m, i):
    return m[i:] + m[:i]

# Uses s_box to compress 48 bits into 32 bits
def s_box_compression(m):
    compressed = []
    for i in range(8):
        # Get bits to determine row
        row = int('0b' + str(m[i * 6]) + str(m[i * 6 + 5]), 2)
        # Get bits to determine column
        col = int('0b' + str(m[i * 6 + 1]) + str(m[i * 6 + 2]) + str(m[i * 6 + 3]) + str(m[i * 6 + 4]), 2)

        index = row * 16 + col
        four = bin(s_box[i][index])[2:].zfill(4)

        for i in four:
            compressed.append(i)

    return compressed

# A bit-wise XOR function for two same-length lists of bits
def XOR(l1, l2):
    xor = []
for i in range(min(len(l1),len(l2))):
    xor.append(int(l1[i]) ^ int(l2[i]))
return xor

def permute(list1, perm_map):
    permuted = []
    for i in range(len(perm_map)):
        permuted.append(list1[perm_map[i]-1])
    return permuted

def split(list1):
    left = []
    right = []
    for i in range(len(list1)/2):
        left.append(list1[i])
        right.append(list1[i+(len(list1)/2)])
    return left,right

def merge(half1, half2):
    merged = []
    for i in range(len(half1)):
        merged.append(half1[i])
    for i in range(len(half2)):
        merged.append(half2[i])
    return merged

def copy(list1):
    list2 = []
    for i in range(len(list1)):
        list2.append(list1[i])
return list2

# Table for initial permutation
init_perm = [58, 50, 42, 34, 26, 18, 10, 2, 60, 52, 44, 36, 28, 20, 12, 4, 62, 54, 46, 38, 30, 22, 14, 6, 64, 56, 48, 40, 32, 24, 16, 8, 57, 49, 41, 33, 25, 17, 9, 1, 59, 51, 43, 35, 27, 19, 11, 3, 61, 53, 45, 37, 29, 21, 13, 5, 63, 55, 47, 39, 31, 23, 15, 7]  

# Inverse table of initial permutation
init_perm_inv = [40, 8, 48, 16, 56, 24, 64, 32, 39, 7, 47, 15, 55, 23, 63, 31, 38, 6, 46, 14, 54, 22, 62, 30, 37, 5, 45, 13, 53, 21, 61, 29, 36, 4, 44, 12, 52, 20, 60, 28, 35, 3, 43, 11, 51, 19, 59, 27, 34, 2, 42, 10, 50, 18, 58, 26, 33, 1, 41, 9, 49, 17, 57, 25]

# Expansion Permutation table that expands 32-bit chunk to 48-bit chunk
exp_perm = [32, 1, 2, 3, 4, 5, 4, 5, 6, 7, 8, 9, 8, 9, 10, 11, 12, 13, 12, 13, 14, 15, 16, 17, 16, 17, 18, 19, 12, 20, 21, 20, 21, 22, 23, 24, 25, 24, 25, 26, 27, 28, 29, 28, 29, 30, 31, 32, 1]

# Lots of S-Boxes
s_box = [
    # SBox 1
    [14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7, 0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8, 4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0, 15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13],
    # SBox 2
    [15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10, 3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5, 0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15]  
]  

29
13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9],
# SBox 3
[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8,
13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1,
13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7,
1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12],
# SBox 4
[7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15,
13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9,
10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4,
3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14],
# SBox 5
[2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9,
14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6,
4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14,
11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3],
# SBox 6
[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11,
10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8,
9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6,
4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13],
# SBox 7
[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1,
13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6,
1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2,
6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12],
# SBox 8
[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7,
1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2,
7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8,
2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]
p_box = [16, 7, 20, 21, 29, 12, 28, 17, 1, 15, 23, 26, 5, 18, 31, 10, 2, 8, 24, 14, 32, 27, 3, 9, 19, 13, 30, 6, 22, 11, 4, 25]

# The following tables are for the key

# Table for the initial key compression
# 64 bits to 56 bits
key_init = [
    57, 49, 41, 33, 25, 17, 9, 1, 58, 50, 42, 34, 26, 18, 10, 2, 59, 51, 43, 35, 27, 19, 11, 3, 60, 52, 44, 36, 63, 55, 47, 39, 31, 23, 15, 7, 62, 54, 46, 38, 30, 22, 14, 6, 61, 53, 45, 37, 29, 21, 13, 5, 28, 20, 12, 4]

# Table of bits to shift by
key_shift = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]

# Compresses key from 56 to 48 bits
key_comp = [
    14, 17, 11, 24, 1, 5, 3, 28, 15, 6, 21, 10, 23, 19, 12, 4, 26, 8, 16, 7, 27, 20, 13, 2, 41, 52, 31, 37, 47, 55, 30, 40, 51, 45, 33, 48, 44, 49, 39, 56, 34, 53, 46, 42, 50, 36, 29, 32]

key_schedule = []

# Methods for printing different states of encryption
def hex_to_bin(hex_num):
    bin_num = []
    for bit in bin(int(hex_num,16))[2:]:
        bin_num.append(bit)
    while len(bin_num)<64:
        bin_num.insert(0,'0')
    return bin_num

def bin_to_hex(bin_list):
    bin_str = '0b'
    for i in bin_list:
        bin_str += str(i)
    return hex(int(bin_str,2))

def p_b(bin_list):
    bin_str = ''
    for i in bin_list:
        bin_str += str(i)
    return bin_str

if __name__ == '__main__':
    plaintext = '0x123456ABCD132536'
    key = '0xAABB09182736CCDD'

    p = hex_to_bin(plaintext)
    k = hex_to_bin(key)

    k = permute(k, key_init)

    ciphertext = DES(p,k)
print bin_to_hex(ciphertext)

plaintext = DES_inv(ciphertext, k)

print bin_to_hex(plaintext)

A.2 RC4

# rc4.py
# Shelley Kandola
# RC4 Implementation

def rc4(pt, k):
    keystream = []
    ciphertext = []

    # Initializing the Sbox
    sbox = [i for i in range(256)]

    # Populating the Sbox
    j=0
    key_list = hex_to_list(k)
    for i in range(256):
        j = (j + sbox[i] + int(key_list[i%len(key_list)],16)) % 256
        temp = sbox[i]
        sbox[i] = sbox[j]
        sbox[j] = temp

    # Generating the Keystream
    i=0
j=0
for c in plaintext:
    i = (i+1) % 256
    j = (j + sbox[i]) % 256
    temp = sbox[i]
    sbox[i] = sbox[j]
    sbox[j] = temp
    t = (sbox[i] + sbox[j]) % 256
    for b in bin(t)[2:].zfill(6):
        keystream.append(b)
    ciphertext = XOR(pt, keystream)
    #print ciphertext
return bin_to_hex(ciphertext)

####### Useful Functions #######
# A bit-wise XOR function for two same-length lists of bits
def XOR(l1, l2):
    xor = []
    # print min(len(l1),len(l2))
    for i in range(min(len(l1),len(l2))):
        xor.append(int(l1[i]) ^ int(l2[i]))
    return xor

# Takes a hex string and makes an list of bits
def hex_to_bin(plaintext):
    bin_text = []
    for hex_chr in plaintext[2:]:
        bits = bin(int(hex_chr,16))[2:].zfill(4)
        for b in bits:
            bin_text.append(b)
return bin_text

# Takes a list of bits and outputs a hex string
def bin_to_hex(bin_list):
    hex_str = ""
    for byte in range(len(bin_list)/4):
        char = '0b'
        for bit in range(4):
            char += str(bin_list[byte*4 + bit])
        hex_str += hex(int(char,2))[2:]
    return '0x'+hex_str

# A nicer way of displaying a list of bits
def p_b(bin_list):
    bin_str = ","
    for i in bin_list:
        bin_str += str(i)
    return bin_str

# Turns a hex string into a list of hex chars
def hex_to_list(hex_string):
    hex_list = []
    for h in hex_string[2:]:
        hex_list.append('0x'+h)
    return hex_list

##############################

if __name__ == "__main__":
    35
plaintext = hex_to_bin('0xABCDe')
print p_b(plaintext)
key = hex_to_bin('0x1234ABCD')
ciphertext = rc4(plaintext, key)
print ciphertext
print rc4(hex_to_bin(ciphertext), key)

A.3 SHA-1

# sha1.py
# Shelley Kandola
# SHA1 One-Way Hash Encryption
# Crypto SYE, Spring 2013

import hashlib

# Takes an int, inverts it
# and returns it as hex
def bin_not(integer):
    ones = 4294967295
    return integer ^ ones

# Takes an integer
# Converts it to bits
# Rotates it, and returns integer
def rotl(bin_string, bit):
    # Ensure string is 32 bits long
    bit_string = bin_string[2:].zfill(32)
    # rotate the string
rotated = bit_string[bit:] + bit_string[:bit]
# add binary prefix
rotated = '0b' + rotated
# convert to hex
# print rotated
rotated = '0x' + hex(int(rotated, 2))[2:].zfill(8)
# return from Long
if rotated[len(rotated)-1]=='L':
    rotated = rotated[:len(rotated)-1]
return rotated

# Input message as hex string
def sha1(message):
    print message
    # Initialize variables:
h0 = '0x67452301'
h1 = '0xEFCDAB89'
h2 = '0x98BADCFE'
h3 = '0x10325476'
h4 = '0xC3D2E1F0'

    ###### Pre-processing:
    # Convert the length of the message
    ### from int to bits to hex
    bin_length = len(bin(int(message, 16))[2:])
    # print "1: ", bin_length
    # Make it a multiple of 8
    bin_length = bin_length + 8 - (bin_length%8)
    # print "2: ", bin_length
    hex_length = hex(bin_length)[2:]
    # print "3: ", hex_length
while len(hex_length)<16:
    hex_length = '0' + hex_length
# print "4: ", hex_length

# Append a 1 to the message
message += '8'

# Pad with 0's until 448%512 bits (112%128 hexadigits)
while ((len(message)-2)%128) != 112: # don't include '0x' in count
    message += '0'

# Add the length of the original message, as 64 bits/16 hexadigits
message += hex_length

##### Process the message in successive 512-bit chunks:
chunks = []
# strip '0x' from suffix
message = message[2:]
# break message into 512-bit (128-hexadigit) chunks
for i in range(len(message) / 128):
    chunk = message[128*i:128*(i+1)]
    words = ['' for index in range(80)]
    chunks.append(words)

    # Extend 16 32-bit words into 80 32-bit words
    for j in range(16):
        chunks[i][j]='0x' + chunk[8*j:8*(j+1)]
    for j in range(16,80):
        c1 = int(chunks[i][j-3],16)
c2 = int(chunks[i][j-8],16)
c3 = int(chunks[i][j-14],16)
c4 = int(chunks[i][j-16],16)

# w[i] = (w[i-3] xor w[i-8] xor w[i-14] xor w[i-16]) leftrotate 1
word = c1^c2^c3^c4
word = rotl(bin(word),1)
chunks[i][j]=word

print chunks

#Initialize hash value for this chunk:
a = int(h0,16)
b = int(h1,16)
c = int(h2,16)
d = int(h3,16)
e = int(h4,16)

for r in range(80):
    if 0 <= r <= 19:
        f,k = (b & c) | (bin_not(b) & d),'0x5A827999'
        k = '0x5A827999'
    elif 20 <= r <= 39:
        f = b ^ c ^ d
        k = '0x6ED9EBA1'
    elif 40 <= r <= 59:
        f = (b & c) | (b & d) | (c & d)
        k = '0x8F1BBCDC'
    elif 60 <= r <= 79:
        f = b ^ c ^ d

39
k = '0xCA62C1D6'
arot5 = rotl(bin(a),5)
temp = (int(arot5,16) + f + e + int(k,16) + int(chunks[i][r],16)) % 2**32
e = d
d = c
c = int(rotl(bin(b),30),16)
b = a
a = temp

# Checking for Test Vector
print r, ": ", hex(a),hex(b),hex(c),hex(d),hex(e)

# Add this chunk's hash to result so far:
h0 = hex((int(h0,16) + a)% 2**32)[2:].zfill(8)
h1 = hex((int(h1,16) + b)% 2**32)
h2 = hex((int(h2,16) + c)% 2**32)
h3 = hex((int(h3,16) + d)% 2**32)
h4 = hex((int(h4,16) + e)% 2**32)
return str(h0)[:10]+str(h1)[2:10]+str(h2)[2:10]+str(h3)[2:10]+str(h4)[2:10]

if __name__=='__main__':
    print sha1('0x'+'abc'.encode("hex"))
m = hashlib.sha1('abc')
print '0x'+m.hexdigest()

    # output should be: aaf4c61ddc5e8a2dabede0f3b482cd9aea9434d

A.4 RSA - Python
A.4.1 Prime Generation

# prime_gen.py

import math,random,time
# Finds two probable n-digit primes
def get_2_primes(n):
    small_primes = sieve(65000)
    p1, p2 = None, None
    while p1 == None:
        p1 = get_prime(small_primes, n)
    while p2 == None:
        p2 = get_prime(small_primes, n)
    return p1, p2

# Finds a probable n-digit prime
def get_prime(small_primes, n):

    lower = 10**(n-1)
    upper = 10**n - 1
    k = 10000

    r = random.randint(lower, upper-k)
    if r % 2 == 0:
        r = r - 1
    t = r + k

    primes = {}
    i = r
    while i < t:
        primes[i] = True
        i = i + 2

    for p in small_primes:
        m = r / p * p
while m < t:
    if m in primes:
        del primes[m]
    m = m + p

for p in primes:
    if approx_prime(p):
        return p

# Lists all primes between 3 and upper with Atkins sieve
def sieve(upper):
    # Search Limit
    limit = upper

    # Initialize the sieve
    is_prime = {}
    i=5
    while i<= limit:
        is_prime[i] = False
        i = i+1

    # Put in candidate primes
    # integers which have an odd number
    # of representations by certain quadratic forms
    sq = int(math.sqrt(limit))
    for x in range(1, sq):
        for y in range(1,sq):
            x2 = x*x
            y2 = y*y

            n = 4*x2 + y2
if (n <= limit) and (n%12==1 or n%12==5):
    is_prime[n] = not is_prime[n]

n = 3*x2 + y2
if (n <= limit) and (n%12==7):
    is_prime[n] = not is_prime[n]

n = 3*x2 - y2
if (x>y) and (n <= limit) and (n%12==11):
    is_prime[n] = not is_prime[n]

# eliminate composites by sieving
for n in range(5,sq):
    n2 = n*n
    if is_prime[n]:
        for k in range(n2,limit,n2):
            is_prime[k] = False

primes = [3]
for p in is_prime:
    if is_prime[p]:
        primes.append(p)
return primes

# Fermat prime test - correct with prob=1/(2^k)
def approx_prime(n):
    k = 100
    while k>0:
        a_i = random.randint(1, n-1)
        if mod_pow(a_i, n-1, n) != 1:
            return False
k=k-1
return True

# Computes x^y mod n
# Iterative version
def mod_pow(x,y,n):
    result = 1
    while y > 0:
        if y%2 == 1:
            result = (result*x) % n
        y /= 2
        x = (x*x) % n
    return result

# Testing and timing
if __name__=='__main__':
    n = 309 # 1024 bits

    print "Direct version:
    a = time.clock()
    lower = 10**(n-1)
    upper = 10**n - 1
    p1 = random.randint(lower, upper)
    while not approx_prime(p1):
        p1 = random.randint(lower, upper)
    p2 = random.randint(lower, upper)
    while not approx_prime(p2):
        p2 = random.randint(lower, upper)
    print n,"digits", time.clock()-a,"seconds"
print "Sieve version:"
a = time.clock()
p1,p2 = get_2_primes(n)
print n,"digits", time.clock()-a,"seconds"

A.4.2 RSA Helper

# RSA_helper.py
# Shelley Kandola
# Enter a bit length and get:
#  public, private keys for RSA

from prime_gen import *

# Computes gcd for a >= b >= 0
# Iterative version
def gcd(a,b):
    while b != 0:
        t = b
        b = a % t
        a = t
    return a

# Generates a number relatively prime to its argument
def relprime(n):
    m = random.randint(2,n-1)
    while gcd(n,m) != 1:
        m = random.randint(2,n-1)
    return m

# Euclid’s Extended Algorithm
# Iterative version
def extended_gcd(a,b):
    x,lastx = 0,1
    y,lasty = 1,0
    while b != 0:
        q = a//b
        a,b = b,a%b
        x,lastx = lastx-q*x,x
        y,lasty = lasty-q*y,y
    return lastx,lasty

# Computes a modular inverse
def inverse(a,m):
    inv = extended_gcd(a,m)[0]
    if inv<0:
        return inv+m
    else:
        return inv

# Computes an RSA keyset
def get_keys(length):
    p1,p2 = get_2_primes(length)
    n = p1*p2
    print "n =",n

    phin = (p1-1)*(p2-1)
    if gcd(3,phin)==1:
        e=3
    else:
        e = relprime(phin)
    print "e =",e
d = inverse(e,phin)
print "d =",d

return n,e,d

A.4.3 RSA

# RSA.py
# Shelley Kandola
# Encrypts then decrypts a message

from RSA_helper import *
import time
# User interface
if __name__=='__main__':

    length = input("Enter a digit length for your prime numbers: ")
a = time.clock()
n,e,d = get_keys(length)
print time.clock()-a

    limit = int(0.83*length)
print "You may enter messages up to", limit, "characters long."
message = raw_input("Enter a string to encrypt: ")

if len(message) > limit:
    print "Message too long."

else:
    hex_m = '0x'+message.encode("hex")
print hex_m
hex_i = int(hex_m, 16)
print hex_i
enc = mod_pow(hex_i, e, n)

dec_i = mod_pow(enc, d, n)
dec_h = hex(dec_i)
chop = dec_h[2:-1]
dec = chop.decode("hex")

print "Encrypted:", enc
print "Decrypted:", dec

A.5 RSA - Java
A.5.1 Key Generation

import java.math.BigInteger;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Random;

public class KeyGen {

    public static void main(String[] args){
        KeyGen rsa = new KeyGen(20);
        rsa.getKeys();
    }

    // Instance Data: the keys
    BigInteger p1;
    BigInteger p2;
    BigInteger n;
    BigInteger e;
    BigInteger d;

}
// Prints your public keys
public void getKeys(){
    System.out.println(n);
    System.out.println(e);
    System.out.println(d);
}

public BigInteger getN(){
    return n;
}

public BigInteger getE(){
    return e;
}

public BigInteger getD(){
    return d;
}

// Constructor: sets instance data
public KeyGen(int digits){
    ArrayList<BigInteger> smallPrimes = sieve(BigInteger.valueOf(65000));
    BigInteger p,q;
    do{p = getPrime(smallPrimes, digits);}
        while(p.equals(BigInteger.ZERO));
    do{q = getPrime(smallPrimes, digits);}
        while(q.equals(BigInteger.ZERO));

    this.p1 = p;
    this.p2 = q;
this.n = p.multiply(q);
BigInteger phin = p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));
if(phin.gcd(BigInteger.valueOf(3))==BigInteger.ONE){
    this.e = BigInteger.valueOf(3);
} else{
    this.e = RSAMath.relprime(phin);
}
//this.d = e.modInverse(n);
this.d = RSAMath.inverse(e,phin);

// Generate a BigInteger between lower and upper
public static BigInteger randBigInt(BigInteger lower, BigInteger upper){
    int bits = upper.bitLength();
    BigInteger result;
    Random rand = new Random();
    do{
        result = new BigInteger(bits, rand);
    }while(result.compareTo(lower)!=1 || upper.compareTo(result)!=1);
    return result;
}

// Generates 1 prime number
public BigInteger getPrime(ArrayList<BigInteger> smallPrimes, int digits){
    // Set upper, lower bound and range
    BigInteger lower = BigInteger.TEN.pow(digits-1);
    BigInteger upper = BigInteger.TEN.pow(digits).subtract(BigInteger.ONE);
    BigInteger k = BigInteger.TEN.pow(4);

    // Determine our random range of numbers to check
    BigInteger r = randBigInt(lower,upper.subtract(k));
```java
if (!r.testBit(0)){
    r = r.subtract(BigInteger.ONE);
}

// Claim everything is prime until we prove it isn't
HashMap<BigInteger, Boolean> primes = new HashMap<BigInteger, Boolean>();
BigInteger t = r.add(k);
BigInteger i = r;
while(i.compareTo(t)==-1){
    primes.put(i, true);
    i = i.add(BigInteger.ONE.add(BigInteger.ONE));
}

// Remove all multiples of primes
for(BigInteger p : smallPrimes){
    BigInteger m = r.divide(p).multiply(p);
    while (m.compareTo(t)==-1){
        if(primes.containsKey(m)){
            primes.remove(m);
        }
        m = m.add(p);
    }
}

BigInteger result = BigInteger.ZERO;
for(BigInteger p : primes.keySet()){  
    if(approxPrime(p)){
        result = p;
        return result;
    }
}
```

private static boolean approxPrime(BigInteger p) {
    Random rand = new Random();

    int k = 100;
    while(k>0)
    {
        BigInteger a = randBigInt(BigInteger.ZERO,p);
        if(mod_pow_big(a,p.subtract(BigInteger.ONE),p).compareTo(BigInteger.ONE)!=0){
            return false;
        }
        k = k - 1;
    }
    return true;
}

private static BigInteger randomBigInt(BigInteger max){
    Random rand = new Random();
    BigInteger r;
    do {
        r = new BigInteger(max.bitLength(), rand);
    } while (r.compareTo(max) >= 0);
    return r;
}

// Modular exponentiation with BigIntegers
private static BigInteger mod_pow_big(BigInteger x, BigInteger y, BigInteger n) {
    if(y.equals(BigInteger.ZERO)){

BigInteger z = mod_pow_big(x, y.shiftRight(1), n);
if(!y.testBit(0)){
    return z.multiply(z).mod(n);
} else{
    return x.multiply(z).multiply(z).mod(n);
}

// Sieve of Atkin for generating primes
private ArrayList<BigInteger> sieve(BigInteger upper){
    BigInteger limit = upper;
    HashMap<BigInteger,Boolean> isPrime = new HashMap<BigInteger,Boolean>(upper.intValue());
    BigInteger i = BigInteger.TEN.shiftRight(1);
    while(i.compareTo(limit)!=1){
        isPrime.put(i, false);
        i = i.add(BigInteger.ONE);
    }
    double sq = Math.sqrt(limit.intValue());
    for(int x=1;x<=sq;x++){
        for(int y=1;y<=sq;y++){
            BigInteger x2 = BigInteger.valueOf(x*x);
            BigInteger y2 = BigInteger.valueOf(y*y);
            BigInteger n = x2.shiftLeft(2).add(y2);
            if ((n.compareTo(limit)!=1) &&
                ((n.mod(BigInteger.valueOf(12)).compareTo(BigInteger.ONE)==0)||
                (n.mod(BigInteger.valueOf(12)).compareTo(BigInteger.TEN.shiftRight(1))

    return BigInteger.ONE;
}
isPrime.put(n, !isPrime.get(n));
}

n = x2.multiply(BigInteger.valueOf(3)).add(y2);

if ((n.compareTo(limit)!=1) &&
   (n.mod(BigInteger.valueOf(12)).compareTo(BigInteger.valueOf(7))==0)){
    isPrime.put(n, !isPrime.get(n));
}

n = x2.multiply(BigInteger.valueOf(3)).subtract(y2);
if ((x>y) && (n.compareTo(limit)!=1) &&
   (n.mod(BigInteger.valueOf(12)).compareTo(BigInteger.valueOf(11))==0)){
    isPrime.put(n, !isPrime.get(n));
}

for(int m=5;m<sq;m++){
    int n2 = m*m;
    if (isPrime.get(BigInteger.valueOf(m))){
        for(int k=n2;k<=limit.intValue();k=k+n2){
            isPrime.put(BigInteger.valueOf(k), false);
        }
    }
}

ArrayList<BigInteger> primes = new ArrayList<BigInteger>();
primes.add(BigInteger.valueOf(3));

for(BigInteger p : isPrime.keySet()){ if(isPrime.get(p)){

54
primes.add(p);
}

return primes;
}

A.5.2 RSA Math

import java.math.BigInteger;
import java.util.Random;

public class RSAMath {

    public static int gcd(int a, int b){
        while(b!=0){
            int t = b;
            b = a%t;
            a=t;
        }
        return a;
    }

    // Generate a number relatively prime to n
    public static BigInteger relprime(BigInteger n){
        Random rand = new Random();
        BigInteger e;
        do{
            e = new BigInteger(n.bitLength(), rand);
        } while(!e.gcd(n).equals(BigInteger.ONE));
        return e;
    }
}
// The extended Euclidean algorithm
public static BigInteger extendedGCD(BigInteger a, BigInteger b){
    BigInteger x = BigInteger.ZERO;
    BigInteger lastx = BigInteger.ONE;
    BigInteger y = BigInteger.ONE;
    BigInteger lasty = BigInteger.ZERO;

    while(!b.equals(BigInteger.ZERO)){
        BigInteger q = a.divide(b);
        BigInteger tmpa = a;
        a = new BigInteger(b.toString());
        b = tmpa.mod(b);

        BigInteger tmpx = x;
        x = lastx.subtract(q.multiply(x));
        lastx = new BigInteger(tmpx.toString());

        BigInteger tmpy = y;
        y = lasty.subtract(q.multiply(y));
        lasty = new BigInteger(tmpy.toString());
    }
    return lastx;
}

// Finds a multiplicative inverse
public static BigInteger inverse(BigInteger a,BigInteger m){
    BigInteger inv = extendedGCD(a,m);
    if((inv.compareTo(BigInteger.ZERO)==-1)){
        return inv.add(m);
    }else{
        return inv;
    }
}
import java.math.BigInteger;

public class RSA {
    public static void main(String[] args) {
        String message = "Hi";
        byte[] mb = message.getBytes();

        BigInteger mbi = new BigInteger(mb);

        System.out.println(mbi);

        BigInteger n = rsa.getN();
        BigInteger e = rsa.getE();
        BigInteger d = rsa.getD();

        System.out.println(n + " +e" + d);

        BigInteger enc = mbi.modPow(e, n);

        BigInteger dec = enc.modPow(d, n);
        System.out.println(dec);

        byte[] db = dec.toByteArray();

        String result = new String(db);
    }
}
System.out.println(result);
}
}

References


