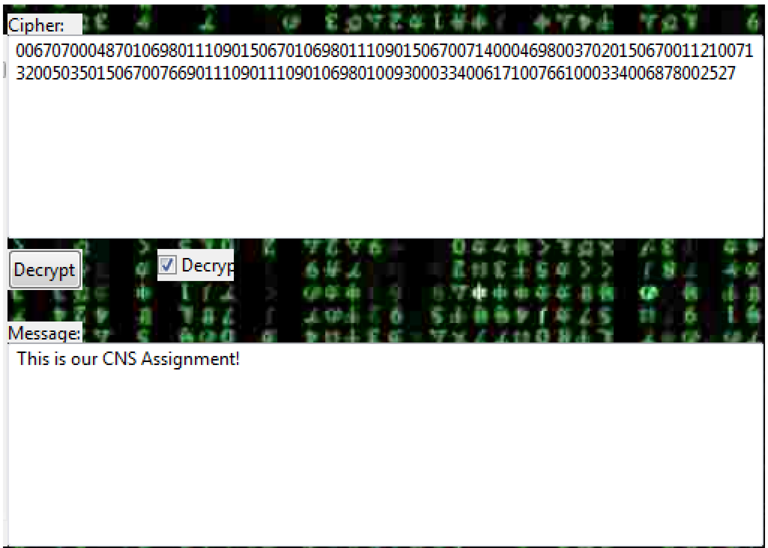
November 28, 2012

RV College of Engineering

Department of Computer Science



Schmidt-Samoa Cryptosystem

By:
Samir Sheriff [1RV09CS093]
Satvik N [1RV09CS095]

1 Introduction

Public-key cryptography refers to a cryptographic system requiring two separate keys, one of which is secret and one of which is public. Although different, the two parts of the key pair are mathematically linked. One key locks or encrypts the plaintext, and the other unlocks or decrypts the ciphertext. Neither key can perform both functions. One of these keys is published or public, while the other is kept private.

How Pubic-key Cryptosystems Work

The distinguishing technique used in public-key cryptography is the use of asymmetric key algorithms, where the key used to encrypt a message is not the same as the key used to decrypt it. Each user has a pair of cryptographic keys - a public encryption key and a private decryption key. The publicly available encrypting-key is distributed, while the private decrypting-key is kept secret. Messages are encrypted with the recipient's public key, and can be decrypted only with the corresponding private key. The keys are related mathematically, but the parameters are chosen so that determining the private key from the public key is either impossible or prohibitively expensive.

Schmidt-Samoa Public-key Cryptosystem

The Schmidt-Samoa cryptosystem is an asymmetric cryptographic technique, whose security, like Rabin and RSA depends on the difficulty of integer factorization.

- Key generation
  - Choose two large distinct primes p and q and compute $N = p^2 \times q$
  - Compute $d = N - 1 \text{ mod } lcm(p-1, q-1)$
  - Now N is the public key and d is the private key.
- **Encryption** - To encrypt a message m we compute the cipher text as $c = m^N modN$
- **Decryption** To decrypt a cipher text c we compute the plaintext as $m = c^d mod(p \times q)$ which like for Rabin and RSA can be computed with the Chinese remainder theorem.
- **Security** - The algorithm, like Rabin, is based on the difficulty of factoring the modulus N, which is a distinct advantage over RSA. That is, it can be shown that if there exists an algorithm that can decrypt arbitrary messages, then this algorithm can be used to factor N.

2 Previous Cryptosystems

RSA Cryptosystem

RSA stands for Ron Rivest, Adi Shamir and Leonard Adleman, who first publicly described it in 1977.

- **Key Generation**
  - Let N = pq be a product of two prime numbers
  - Compute $(n) = (p1)(q1)$, where is Euler's totient function.

- Choose an integer e such that 1 ¡ e ¡ (n) and greatest common divisor of (e, (n)) = 1; i.e., e and (n) are coprime.
- Determine d as: d e-1 (mod (n)) i.e., d is the multiplicative inverse of e mod (n).

- **Encryption**: Let M be a message, and c the ciphertext. Then, $c = m^e (modn)$
- **Decryption**: $m = c^d (modn)$ By construction, d*e= 1 mod (n). The public key consists of the modulus n and the public (or encryption) exponent e. The private key consists of the modulus n and the private (or decryption) exponent d which must be kept secret.

Rabins Cryptosystem

In 1979, Michael Rabin suggested a variant of RSA with public-key exponent 2, which he showed to be as secure as factoring.

- **Key Generation**
  - Choose two large distinct primes p and q.
  - Let n=pq. Then n is the public key. The primes p and q are the private key
- **Encryption**: For the encryption, only the public key n is used. The process follows - Let P = {0,...,n-1} be the plaintext space (consisting of numbers) and $m \in P$ be the plaintext. Now the ciphertext is determined by $c = m^2$(mod n).

  c is the quadratic remainder of the square of the plaintext, modulo the key-number n.
- **Decryption**: To decode the ciphertext, the private keys are necessary. The process follows: If c and r are known, the plaintext is then $m \in \{0,...,n-1\}$ with $m^2 = c$ (mod r). For a composite r (that is, like the Rabin algorithm's) there is no efficient method known for the finding of m. If, however r is prime (as are p and q in the Rabin algorithm), the Chinese remainder theorem can be applied to solve for m.

  Thus the square roots $m_p = \sqrt{c}$ mod p and $m_q = \sqrt{c}$ mod q must be calculated

3 Implementation

```java
package com.jinkchak;

import java.security.InvalidAlgorithmParameterException;

import org.eclipse.swt.widgets.Display;

public class Schmidt_Samoa_Encryptor {
        private int p, q;
        private int public_key, private_key;

        private static final int BLOCK_SIZE = 6;   //For splitting a String of text into blocks

        /**
         * This constructor initializes the following variables:
         *   p - with a default value of 23
         *   q - with a default value of 31
         * After that, it calls a method that computes the private and public keys
```

```
     *
     */
    public Schmidt_Samoa_Encryptor()
    {
        reInitialize(23, 31);
    }

    /**
     * Re−initializes the system with the new values for p and q, and then
     * computes the new values of the public and private keys.
     * @param p A large prime number
     * @param q A large prime number that is distinct from q
     *
     */
    public void reInitialize(int p, int q)
    {
        this.p = p;
        this.q = q;
        public_key = computeN();
        try {
            private_key = modular_Equation_Solver(public_key, 1, lcm(p−1, q−1));
        } catch (InvalidAlgorithmParameterException e) {
            e.printStackTrace();
        }

    }

    /**
     * Computes the lcm of two integers
     * @param a An integer
     * @param b An integer
     * @return LCM of a and b
     */
    private int lcm(int a, int b)
    {
        return (a*b)/gcd(a,b);
    }

    /**
     * Computes the GCD of two integers
     * @param a An integer
     * @param b An integer
     * @return GCD of a and b
     */
    private int gcd(int a, int b) {
        if (b==0)
            return a;
        return gcd(b,a%b);
    }

    /**This method contains an implementation of the extended Euclidean algorithm.
     * The extended Euclidean algorithm is an extension to the Euclidean algorithm.
     * Besides finding the greatest common divisor of two integers, as the Euclidean algorithm does,
     * it also finds integers x and y (one of which is typically negative) that satisfy Bzout's identity:
```

```
	 *    ax + by = gcd(a, b)
	 * @param a An integer
	 * @param b An integer
	 * @return An integer array z consisting of three element:
	 *    z[0] = gcd(a, b)
	 *    z[1] = x
	 *    z[2] = y
	 */
	private int[] extendedEuclidsAlgo(int a, int b)
	{
		int []result = new int[3];
		if(b==0)
		{
			result[0] = a;// index 0 is x
			result[1] = 1;// index 1 is y
			result[2] = 0;// index 2 is d ... ax+by = d
			//System.out.println(result[0]+" "+result[1]+" "+result[2]);
			return result;
		}

		int []result_temp = extendedEuclidsAlgo(b, a%b);
		int []final_result = {result_temp[0],result_temp[2],result_temp[1]-(a/b)*result_temp[2]};
		//System.out.println(final_result[0]+" "+final_result[1]+" "+final_result[2]);
		return final_result;
	}

	/**
	 * This method implements the modular exponentiation algorithm as defined in the CLRS text book.
	 * It finds out the result of (a^b) mod n, even when b is very very large
	 * @param a An integer that has to be raised to the power b
	 * @param b An integer that denotes the power to which a has to be raised.
	 * @param n An integer based on which all multiplication operations are performed (mod n)
	 * @return An integer containing the result of ((a ^ b) mod n)
	 */
	public int modularExponentiator(int a, int b, int n)
	{
		int c = 0;
		int d = 1;
		String binaryB = Integer.toBinaryString(b);

		for(int i = 0; i < binaryB.length(); i++)
		{
			c = 2 * c;
			d = (d * d) % n;
			if(binaryB.charAt(i) == '1')
			{
				c++;
				d = (d * a) % n;
			}
		}

		return d;
	}

```

```
    /**
     * Encrypts a message using the Schmidt−Samoa Algorithm. The message is split into blocks of size
     * BLOCK_SIZE and each block is encrypted to form a cipher string. If a given block is less than the
     * BLOCK_SIZE, then the toNLengthString() method is called to
     * convert the block to a string of size BLOCK_SIZE.
     * @param message A string of plaintext.
     * @return A string containing the cipher text
     */
    public String encrypt(String message)
    {
        int []cipher = new int[message.length()];
        String cipherString = "";
        for(int i = 0; i < message.length(); i++)
        {
            cipher[i] = encrypt(message.charAt(i));
            cipherString += toNLengthString("" + cipher[i], BLOCK_SIZE);
        }

        System.out.println("STRING = " + cipherString);
        return cipherString;
    }

    /**
     * This method encrypts only an integer.
     * It is used by the encrypt(String) method on each block of the plaintext *
     * @param m An integer that has to be encrypted
     * @return An integer containing an encrypted version of m, i.e., ((m ^ public_key) mod (public_key))
     */
    public int encrypt(int m)
    {
        return modularExponentiator(m, public_key, public_key);
    }

    /**
     * This method decrypts only an integer. It is used by the decrypt(String)
     * method on each block of the ciphertext.
     * @param c An integer that has to be decrypted. It should satisfy the constraint −−−− 0 < M < (p * q)
     * @return An integer containing the decrypted version of c, i.e., ((c ^ private_key) mod(p * q))
     */
    public int decrypt(int c)
    {
        return modularExponentiator(c, private_key, p * q);
    }

    /**
     * Decrypts a message using the Schmidt−Samoa Algorithm.
     * The message is split into blocks of size
     * BLOCK_SIZE and each block is decrypted to form a plaintext string.
     * @param message A string of cipher text.
     * @return A string containing the plain text
     */
    public String decrypt(String cipher)
    {
        String plaintext = "";
```

```java
        int [] message = new int[cipher.length()];
        for(int i = 0; i < cipher.length() / BLOCK_SIZE; i++)
        {
            message[i] = Integer.parseInt(cipher.substring(i * BLOCK_SIZE,
                                        i * BLOCK_SIZE + BLOCK_SIZE));
            message[i] = decrypt(message[i]);
            plaintext += (char)message[i];
        }
        return plaintext;
    }

    /**
     * Displays all details of the following values:
     *   p
     *   q
     *   Public Key
     *   Private Key
     * @return A string containing these values
     */
    public String display()
    {
        String message = "Algorithm details \n p = "+p + " q = "+
                    q + "\nPublic Key is "+public_key+
                    "\nPrivate Key is "+private_key+"\n";
        System.out.println(message);
        return message;
    }

}
```

References

[1] Katja Schmidt-Samoa, *A New Rabin-type Trapdoor Permutation Equivalent to Factoring and Its Applications*. TechnischeUniversit. samoa@informatik.tu-darmstadt.de

[2] Schmidt-Samoa Cryptosystem - `http://en.wikipedia.org/wiki/Schmidt-Samoa_cryptosystem`

[3] Rivest, R.; A. Shamir; L. Adleman (1978). *"A Method for Obtaining Digital Signatures and Public-Key Cryptosystems"*. Communications of the ACM 21 (2): 120126. doi:10.1145/359340.359342

[4] Joe Hurd, Blum Integers (1997) - `http://www.gilith.com/research/talks/cambridge1997.pdf`

[5] Rabin, Michael. *Digitalized Signatures and Public-Key Functions as Intractable as Factorization*. MIT Laboratory for Computer Science, January 1979.

[6] Katja Schmidt-Samoa - *Contributions to Provable Security and Efficient Cryptography.* `http://tuprints.ulb.tu-darmstadt.de/708/1/Diss.Schmidt-Samoa.pdf`

4 Screenshots

Figure 1: Encryption using the Schmidt-Samoa algorithm

Figure 2: Decryption using the Schmidt-Samoa algorithm.