Implementing Public-Key Cryptography in Haskell*

***** ****,
**********,
**********,
*********, ****** 9,
*******.

*****@*******.***.***

November 12, 2001

Abstract

In this paper we describe how the RSA public-key cipher can be implemented in the functional programming language *Haskell* and we present a complete working implementation. This implementation serves two useful pedagogical purposes: (1) it is a very clear, high-level implementation of RSA that can easily be understood and manipulated by students of cryptography, and (2) it is an excellent example of how a functional programming language such as *Haskell* can be used to implement *real* applications.

While the main goal of our implementation is clarity rather than efficiency, no treatment of public-key cryptography can ignore efficiency and we show how well-known techniques for improving the efficiency of RSA can be implemented. We also compare the performance of our implementation with an equivalent implementation in *Java*, and show that it is also practical to implement public-key cryptography in *Haskell*.

Keywords: Functional Programming; Haskell; Public-Key Cryptography; RSA

*This research was partially funded by **********.

1 Introduction

Traditionally, for efficiency reasons, cryptographic software has been written in low-level programming languages such as *C* [15] and *C++* [6]. In fact, it is not uncommon for assembly language to be used to implement frequently used routines, e.g., those that perform arithmetic operations on arbitrarily large integer values. However the recent widespread use of cryptographic software [27, 28] written in *Java* [11], has demonstrated that other considerations, in particular portability, can make the use of higher level languages acceptable. There is no doubt that *Java* implementations are less efficient than *C* or *C++* implementations and are therefore of limited use in high-volume servers or devices with limited computing power, but for most client software, portability is far more important than efficiency.

To allow students to implement software as part of a course on public-key cryptography, the author considered clarity and ease of use to be overriding considerations. As a result, three properties of the functional programming language *Haskell* [20, 21] led the author to select it as an implementation language for students:

- *Haskell* has built-in support for arbitrarily large integer values. Since most public-key cryptography is based on the use of large integer values, built-in support makes programming much simpler.
- It is relatively easy to convert the notations used in standard texts [26, 19] on public-key cryptography into *Haskell* code.
- The interactive nature of *Haskell* environments such as *HUGS* [14] and *GHCi* [29] allows students to explore the properties of algorithms in a simple, natural way.

After some initial use, it became clear that using *Haskell* to implement public-key cryptography was not only pedagogically sound, but that *industrial-strength* cryptography could also be implemented. Such implementations can never compete with more traditional implementations in terms of efficiency, but they can be used to add cryptographic functionality to *Haskell* applications. In addition, the clarity of such implementations make them suitable for use as *reference implementations* and *executable specifications*.

In section 2 we look at some basic ideas in cryptography. In section 3 we illustrate how a public-key cipher and associated functionality can be implemented in *Haskell* by exploring an implementation of RSA [22]. In section 4 we present a small example application and give some performance figures for our implementation. Finally, in section 5, we draw some conclusions.

The *Haskell* code used to illustrate our approach is given in figures throughout this paper, but to keep the total length of the paper manageable, we only present a small amount of the code from which all comments have been elided. A complete description of the implementation is presented in an accompanying report [1] which is available together with the source code from the author's WEB site [2].

2 Cryptography

There are three classes of algorithm used in cryptography:

- *Symmetric Ciphers.*
- *Asymmetric Ciphers.*
- *Hash Functions.*

Symmetric ciphers implement classic cryptography in which shared *secret* keys allow entities to encrypt and decrypt data. The main use of symmetric ciphers is to keep information *confidential*, i.e., two users wishing to keep some *plaintext* confidential can encrypt it using a shared key so as to produce *ciphertext*. Provided shared keys are kept secret, an adversary cannot recover the plaintext from the ciphertext[1]. There are numerous symmetric ciphers in use; commonly used ciphers include DES [10], IDEA [17] and Skipjack [8].

Asymmetric or *public-key* ciphers are relatively new techniques first described by Diffie and Hellman in 1976 [4]. With asymmetric cryptography, an entity generates a pair of mathematically related keys and keeps the so-called

[1]Except for a *one-time pad*, all ciphers are susceptible to *cryptanalysis* which allows plaintext to be recovered from ciphertext without the use of keys. However, if *strong* ciphers and sufficiently long keys are used, such *attacks* are *computationally infeasible*.

private-key a secret while publishing the *public-key*. At first sight, asymmetric cryptography looks counterintuitive, however, asymmetric ciphers have two modes[2] of use that can be used to implement a wide range of security services:

- A public-key can be used to encrypt data and thus produce ciphertext that can only be decrypted by the corresponding private-key, i.e., the plaintext can only be recovered by the owner of the *key-pair*.

- A private-key can be used to encrypt data and thus produce ciphertext that can only be decrypted by the corresponding public-key. Of course, since a public-key is intended to be widely known, this does not provide confidentiality. However, if the original plaintext is redundant (e.g., it is known to be a well-formed *Haskell* program), anyone with access to the public-key can recover the plaintext and check this redundancy. If the recovered plaintext is valid, then the ciphertext must have been produced by the private-key. This is the basis of a *digital signature* and can be used to provide *integrity*, *authentication* and *non-repudiation* services.

The most widely used asymmetric cipher is RSA [22]. Other examples include DSA [9], ElGamal [5] and those based on elliptic curves [3].

In practice, asymmetric ciphers are much less efficient than symmetric ciphers, so it is common to use a combination of techniques. For example, many security protocols use asymmetric ciphers to exchange symmetric *session* keys which are then used for encrypting data[3].

Hash functions are used in many areas of computing, e.g., *hash tables* are a widely used data structure that allow for the efficient storage and retrieval of data in an array. A hash function takes an arbitrarily large amount of data and produces a small *hash* or *digest*. Clearly, a hash function is many-to-one, but for use in cryptography we require functions for which it is computationally infeasible to find two pieces of data that produce the same digest. Commonly used hash functions include SHA-1 [7] and RIPEMD-160 [13]; both of which produce 160-bit digests.

[2]Not every asymmetric cipher can be used in both modes.

[3]For example, A can send B a symmetric key encrypted using B's public-key. This approach reduces the complexity of *key management* as symmetric keys can be generated when needed.

Hash functions can be used with asymmetric ciphers to implement efficient *digital signature schemes*. With such a scheme, the data is first hashed and the digest is then encrypted with the signer's private-key to produce the signature. An entity wishing to check a digital signature recreates the digest by hashing the original data and comparing this with the plaintext recovered from the signature using the signer's public-key.

More details can be found in standard textbooks on cryptography [26, 19, 24].

2.1 Trapdoor One-Way Functions

Asymmetric cryptography is based on the use of *trapdoor one-way functions* [19].

A function $f : A \to B$ is *one-way* if there is an efficient algorithm for computing $f(a)$ where $a \in A$, but there is no efficient algorithm for computing the inverse function $f^{-1} : B \to A$, i.e., given $b \in \mathbf{ran}(f)$ it is computationally infeasible to find $a \in A$ such that $f(a) = b$. A trapdoor one-way function is a one-way function in which f^{-1} can be computed efficiently if some additional *trapdoor information* is known.

Typically, an asymmetric cipher consists of a family of trapdoor one-way functions indexed by public-keys. We will consider an asymmetric cipher as consisting of a higher-order function $\mathcal{E}$ such that $\mathcal{E}(K_{pub})$ is the one-way function for the public-keys K_{pub}. Thus, $\mathcal{E}(K_{pub})(m)$ represents the encryption of the plaintext m by the public-key K_{pub}.

There will also be a higher-order function $\mathcal{D}$ such that $\mathcal{D}(K_{priv})$ is the inverse function of $\mathcal{E}(K_{pub})$ if K_{priv} is the private-key corresponding to the public-key K_{pub}. Clearly, we require $\mathcal{D}(K_{priv})(\mathcal{E}(K_{pub})(m)) = m$.

Trapdoor one-way functions are based on mathematical problems that are *believed* to be intractable. Such problems include *integer factorisation* and the *discrete log problem* [19].

3 Implementing RSA

RSA [22] is the most widely known and deployed public-key cryptosystem. Like many public-key cryptosystems, RSA uses modular arithmetic on very large

```
gcde :: Integer → Integer → (Integer,Integer,Integer)

gcde a b =
  let
    gcd_f (r1,x1,y1) (r2,x2,y2)
      | r2 == 0     = (r1,x1,y1)
      | otherwise   =
         let
           q = r1 `div` r2
           r = r1 `mod` r2
         in
           gcd_f (r2,x2,y2) (r,x1−q∗x2,y1−q∗y2)
    (d,x,y) = gcd_f (a,1,0) (b,0,1)
  in
    if   d < 0
    then (−d,−x,−y)
    else (d,x,y)
```

Figure 1: The extended Euclidean algorithm

numbers (typically 300-600 decimal digits).

3.1 Modular Arithmetic in *Haskell*

While *Haskell* has built-in support for arbitrarily large integers and modular arithmetic, some of the operations required for RSA are not supported or have inefficient implementations. In this section we will look at some simple number theory [16, 25] and the implementation of some useful algorithms in *Haskell*.

The set $\mathbb{Z}_m = \{0, 1, .., m-1\}$ of integers *modulo* m forms a *commutative ring* [12] under the normal definitions of addition and multiplication modulo m, i.e., $\mathbb{Z}_m$ forms a *group* with respect to addition and a *semigroup* with respect to multiplication. This means that modular addition and multiplication are commutative and associative, and that multiplication distributes over addition. In addition, each element $a \in \mathbb{Z}_m$ has an additive inverse $(m - a)$ **mod** m, but not every element has a multiplicative inverse $a^{-1} \in \mathbb{Z}_m$ such that $a^{-1}a \equiv$

```
invm :: Integer → Integer → Integer

invm m a
       | g /= 1       = error "No inverse exists"
       | otherwise    = x `mod` m
       where (g,x,_) = gcde a m
```

Figure 2: Inverse modulo M

```
expm :: Integer → Integer → Integer → Integer

expm m b k = (b ^ k) `mod` m
```

Figure 3: Naïve modular exponentiation

1 $(mod\ m)$.

However, it can be shown that $a \in \mathbb{Z}_m$ has a multiplicative inverse if and only if $gcd(a, m) = 1$. Given a and m, the extended Euclidean algorithm [16] can be used to compute the triple (d, x, y) where $d = gcd(a, m)$ and $ax + my = d$. In addition, if $d = 1$, a has an inverse given by x **mod** m. The extended Euclidean algorithm is implemented as the function *gcde* (Figure 1) and the function *invm* (Figure 2) can be used to compute the inverse modulo m if it exists.

If m is prime, $\mathbb{Z}_m$ forms a (finite) *field* with the *multiplicative group* $\mathbb{Z}_m^* = \{1, .., m-1\}$. In particular, for each $a \in \mathbb{Z}_m^*$, we have $gcd(a, m) = 1$ and therefore, each a has a multiplicative inverse.

Haskell supports integer exponentiation and therefore, a naïve implementation of the modular exponentiation $a^b\ (mod\ m)$ would be that given in Figure 3. However, for the integer values used in cryptography this is hopelessly inefficient in terms of both time and space. Fortunately, the well-known *square-and-multiply* algorithm [25] produces significantly faster code and performing modular reduction at each step reduces the size of intermediate results; see Figure 4.

A fuller description of this *Haskell* code is given in [1]. In particular, [1] contains a number of different implementations of modular exponentiation to illustrate the relative efficiency of different approaches and the need for tail

```
expm :: Integer → Integer → Integer → Integer

expm m b k =
   let
     ex a k s
        | k == 0          = s
        | k `mod` 2 == 0  = ((ex (a*a `mod` m)) (k `div` 2)) s
        | otherwise       = ((ex (a*a `mod` m)) (k `div` 2)) (s*a `mod` m)
   in ex b k 1
```

Figure 4: Modular exponentiation

```
data RSAPublicKey  = PUB  Integer Integer -- (n,e)
data RSAPrivateKey = PRIV Integer Integer -- (n,d)
```

Figure 5: RSA keys

recursion and strict arguments in the implementation of *expm*.

3.2 The RSA Algorithm

The *Euler phi* or *Totient* function $\phi(m)$ is defined as the number of integers in the range $1..m$ that are relatively prime to m. We have:

$$\phi(m) = \sharp\{a \mid 1 \leq a \leq m \wedge gcd(a, m) = 1\}$$

In general, for large m computing $\phi(m)$ is computationally infeasible. However, for some special cases, $\phi(m)$ is easy to compute. In particular, given prime p, then $\phi(p) = p - 1$ and given $n = pq$ where p and q are distinct primes, $\phi(n) = (p - 1)(q - 1)$.

It can also be shown that if $x \equiv y \ (mod\ \phi(n))$ then $a^x \equiv a^y \ (mod\ n)$. In particular, if we choose d and e such that $de \equiv 1 \ (mod\ \phi(n))$, then $(x^e)^d \equiv x \ (mod\ n)$.

These two properties form the basis of RSA. If we select two large distinct primes p and q, then we can compute a modulus $n = pq$. If we know p and q, then we can easily compute $\phi(n) = (p - 1)(q - 1)$. However, if we only know n, it is computationally infeasible to factor n and compute $\phi(n)$ or to compute

```
genRSAKey :: Integer → Integer → (RSAPrivateKey,RSAPublicKey)
genRSAKey p q =
    let
      phi = (p–1)*(q–1)
      n   = p*q
      e   = find (phi `div` 5)
      d   = invm phi e
      find x
        | g == 1    = x
        | otherwise = find ((x+1) `mod` phi)
        where (g,_,_) = gcde x phi
    in
      (PRIV n d,PUB n e)
```

Figure 6: RSA key generation

$\phi(n)$ directly.

Given n and $\phi(n)$, we can select an arbitrary *public exponent* e such that $1 < e < \phi(n)$ and $gcd(e, \phi(n)) = 1$. Since $gcd(e, \phi(n)) = 1$, e has an inverse d modulo $\phi(n)$ that we use as the *private exponent*.

Using these exponents, we have a public-key (n, e) and plaintext x such that $0 \leq x < n$ is encrypted using x^e **mod** n. (n, d) is the private-key and ciphertext y such that $0 \leq y < n$ is decrypted using y^d **mod** n. Since $ed \equiv 1\ (mod\ \phi(n))$, encryption followed by decryption gives us $(x^e)^d \equiv x^{ed} \equiv x\ (mod\ n)$.

In *Haskell*, we represent RSA keys as pairs of integers (Figure 5). Given two suitably large distinct primes, we can generate a key-pair using the function *genRSAKey* (Figure 6). In *genRSAKey* we select an arbitrary initial value for the public exponent e and increment this value until $gcd(e, \phi(n)) = 1$. Alternatively, given two suitably large distinct primes and a proposed public exponent, the function *computeRSAKey* (Figure 7) will compute the key-pair or generate an error if the proposed public exponent is not suitable.

Encryption with a public-key and decryption with a private-key are implemented by the functions *ersa* and *drsa* respectively (Figure 8). However, since RSA is reversible, *drsa* can be used to perform encryption with a private-key

```
computeRSAKey :: Integer → Integer → Integer → (RSAPrivateKey,RSAPublicKey)

computeRSAKey p q e =
    let
      phi     = (p–1)*(q–1)
      (g,_,_) = gcde e phi
      n       = p*q
      d       = invm phi e
    in
      if   (g /= 1)
      then error "Public exponent not acceptable"
      else (PRIV n d,PUB n e)
```

Figure 7: RSA key generation with a given public exponent

```
ersa :: PublicKey → Integer → Integer
ersa (n,e) x = expm n x e

drsa :: PrivateKey → Integer → Integer
drsa (n,d) x = expm n x d
```

Figure 8: RSA encryption and decryption

and *ersa* decryption with a public-key. Complete code and examples are given in [1].

3.3 Enhancements

The *Haskell* code presented in sections 3.1 and 3.2 is complete, but somewhat simplistic. In particular, it only handles the encryption and decryption of relatively small amounts of data and we have not addressed the problem of selecting suitable primes p and q. In this section we look at how we can handle the encryption of arbitrary data and some commonly used techniques for improving the efficiency of RSA. In section 3.4 we revisit the issue of selecting primes and generating keys.

```
class Split a where
   split   :: Integer → a → [Integer]
   combine :: Integer → [Integer] → a
```

Figure 9: The split class

```
e_rsa :: (Split a) ⇒ RSAPublicKey → a  → [Integer]
e_rsa k@(PUB n _) x = map (ersa k) (split n x)

d_rsa :: (Split a) ⇒ RSAPrivateKey → [Integer] → a

d_rsa k@(PRIV n _) x          = combine n (map (drsa k) x)
```

Figure 10: Generic RSA encryption and decryption functions

3.3.1 Arbitrary Data

The modulus n of a key-pair (n, e) and (n, d) determines the maximum size of integer that can be encrypted using $ersa$ (n, e) and decrypted using $drsa$ (n, d). To handle larger data (e.g., a string of arbitrary length) it is necessary to *split* the data into a list (say ℓ) of integer values each in the range 0..n–1. The list ℓ can then be encrypted by mapping $ersa\ public$ where *public* is the public-key, to each element[4] in the list to produce a list of ciphertexts (say ℓ_c).

$$\ell_c = map\ (ersa\ public)\ \ell$$

Decryption is the reverse process, i.e., mapping $drsa\ private$ where *private* is the private-key, to each element in the list ℓ_c.

$$\ell' = map\ (drsa\ private)\ \ell_c$$

Clearly, if we use corresponding public and private-keys, we have $\ell = \ell'$.

Rather than define separate encryption and decryption functions for each potential type that must be handled, we define a class *Split* (Figure 9) that cap-

[4]This *mode* of operation is known as *Electronic Codebook* (ECB) mode. ECB has known weaknesses, e.g., depending on how the original data is split, it may be possible to rearrange and/or remove blocks of ciphertext and still have a list of ciphertexts that decrypt to something meaningful. Such weaknesses can be overcome by using a chaining mode such as *Cipher-Block Chaining* (CBC). See [19] for more details.

tures the concept of splitting a value into a list of integers. This class supports two functions, *split* and *combine* such that for any integer $n > 1$ we have two properties:

1. *split* n a segments a into a list of integers each in the range $0..n-1$, i.e., if *split* n $s = \ell$ then for all i such that $0 \leq i < length\ \ell$ we have $0 \leq (\ell\ !!\ i) < n$.

2. *combine* n is the inverse of *split* n, i.e., *combine* n (*split* n a) $= a$ for all a.

With RSA, we use a key's modulus n with *split* and thus all integers in the resulting list can be be encrypted and encrypted with the given key.

Using the class *Split* we can write more generic encryption and decryption functions *e_rsa* and *d_rsa* that can operate on any *Split* type (Figure 10). It is then relatively easy to write functions that allow arbitrarily large integers, strings, etc. to be treated as members of the *Split* class. Full details and examples are given in [1].

3.3.2 The Chinese Remainder Theorem

There are two common approaches used to improve the efficiency of RSA:

1. A small public exponent e can be selected so that exponentiation with e requires fewer steps. In particular, with the *square-and-multiply* algorithm [25], since each `1`-bit requires a multiplication, public exponents with a small number of `1`-bits are typically selected. Typically[5] the values 3 and 65537 [19] are used.

2. The efficiency of exponentiation with the private exponent can be improved by using an algorithm based on the *Chinese Remainder Theorem* (CRT) [25]. Details of this algorithm can be found in [19].

The code presented earlier can be used with small public exponents, i.e., the function *computeRSAKey* can be supplied with a suitable small exponent

[5]The use of the public exponent 3 is subject to an attack [19] if the same message is encrypted for three or more recipients. However, this attack can be avoided by a process known as *salting* in which each recipient is sent a variant of the original plaintext.

```
data RSAPrivateKey = PRIV Integer Integer  -- (n,d)
                   | CRT  Integer Integer Integer Integer Integer Integer Integer
                          -- (n,d,p,q,d mod (p-1),d mod (q-1),(inverse q) mod p)
```

Figure 11: Extended RSA private-keys

and the resulting key can be used with *ersa*, *e_rsa*, *drsa* and *d_rsa*. Of course, *computeRSAKey* will be undefined if the given public exponent is not relatively prime to the computed modulus. As we will see later, this problem can be avoided by selecting suitable primes.

In addition to the modulus n and private exponent d, the CRT algorithm requires the prime factors p and q of n. Therefore, to use this algorithm it is necessary to store p and q as part of the private-key. There are also a number of values $exp1 = d \textbf{ mod } (p-1)$, $exp2 = d \textbf{ mod } (q-1)$ and $coeff = p^{-1} \textbf{ mod } q$ which can be *pre-computed* and these are stored as part of the private-key.

While it is more efficient to use the CRT algorithm, it is not always possible to obtain suitable private-keys. When supplied with only a private exponent d and modulus n, it is computationally infeasible to determine the factors of n, i.e., it is computationally infeasible to compute the extra components needed for the CRT algorithm. As RSA private-keys may be obtained from different sources, it is necessary to support both forms of RSA private-keys and this gives rise to the definition for a private-key given in Figure 11.

The functions used to implement RSA can easily be extended to support both forms of private-key. For example, `drsa` can be extended as shown in Figure 12. The complete implementation of the RSA using the CRT algorithm is given in [1].

3.4 Key Generation

To generate an RSA key-pair, we must select two large distinct primes p and q of roughly the same size[6] and the resulting modulus $n = pq$ will have approximately twice the number of digits as these primes. Clearly, these primes must

[6]Since an adversary need only find one of these primes, security is not increased by having one prime being significantly larger than the other.

```
drsa :: RSAPrivateKey → Integer → Integer

drsa (PRIV n d) x = expm n x d

drsa (CRT n d p q exp1 exp2 coeff) x =
        let
          (a1,a2) = (expm p x exp1,expm q x exp2)
          u = ((a2−a1)∗coeff) `mod` q
        in
          a1 + u∗p
```

Figure 12: The extended `drsa` function

be kept secret by the owner of the key-pair and it must be computationally infeasible for an adversary to guess or compute them.

Given current factorisation techniques, it is recommended that RSA keys should have moduli of at least 1024 bits ($\approx$ 310 decimal digits) and therefore, we need to select two 512-bit primes ($\approx$ 155 decimal digits). There are two major[7] issues that must be addressed:

1. There are approximately 10^{151} primes of 512 or less bits, so provided p and q are selected at random, it is computationally infeasible for an adversary to search for the factors of n. However, care needs to be taken to ensure that the selection process is totally unpredictable, i.e., there must be nothing in the selection process that actually reduces the search space to a subset of values which is sufficiently small to allow an exhaustive search.

2. There are no known efficient algorithms for computing large primes or for determining if a given large integer is prime. There are, however, a number of *probabilistic primality tests* that can be used to determine if a given large integer is probably prime. An example is *Fermat's Test* [19] that can be used to test if a large integer is a prime with an arbitrarily

[7]There are a number of other issues relating to the selection of p and q. For example, some authors consider it important to use *safe* primes; see [19].

small probability of it being composite[8].

Since primes are uniformly distributed, suitable primes can be found by selecting large odd numbers at random and testing if they are prime. Alternatively, we can select a large odd number x at random and use the first value in the sequence $x, x+2, x+4, \ldots$ that is (probably) prime.

Like most programming languages, *Haskell* has library facilities to implement *pseudo-random number generators* that can be used to generate random numbers; see the standard module *Random* [21]. Such generators are typically based on *linear congruential generators* [16] and are designed to produce statistically random numbers. While they are useful in simulation programs, such generators are not *cryptographically secure*.

1. They are *predictable*. Given values produced by such a generator, it is possible to predict future values.

2. From the point of view of cryptography, they repeat after relatively short *periods*.

3. They are initialised (*seeded*) with relatively small values. This reduces the search space for an adversary.

There are various cryptographically secure pseudo-random number generators that are not predictable, have sufficiently long periods and take sufficiently large seed values [19]. Here we consider the *Blum-Blum-Shub* (BBS) pseudo-random bit generator [26]. An instance of this generator produces a cryptographically secure sequence of bits. Internally, the BBS pseudo-random bit generator produces a sequence $x_0, x_1, \ldots$ of numbers such that $x_{i+1} = x_i^2 \ \mathbf{mod}\ n$ where n is the product of two distinct secret primes p and q. The value x_0 is the seed supplied to the generator and the bits generated are the least significant bits of the numbers $x_1, x_2, \ldots$. Thus, to implement BBS, we must store the modulus n and the current value x_i.

One possible implementation strategy would be to implement these pseudo-random number generators as types of the standard class *Random*. However,

[8] Fermat's Test has a number of weaknesses. In particular, when supplied with a *Carmichael Number* [25] there is a high probability that it will fail to detect that it is composite. A more secure approach is the *Miller-Rabin Test* [19].

this was considered inappropriate for the following reasons:

1. The standard class *Random* supports a function *split* that creates two generators from a single generator. The semantics of this function are unclear and it is not known if such splitting is cryptographically sound. In any case, there is no real need for such functionality.

2. Pseudo-random number generators have an internal state that is used to compute the next value. Therefore, it would appear that support for monadic programming [30, 31] would be a worthwhile approach.

3. The functionality[9] required is somewhat different than that supported by the standard class *Random*.

Therefore, we define a class *PRNG* (Figure 13) which captures the basic properties of a pseudo-random number generator. The function *nextB* generates a random bit, *nextI s* generates a random number with s bits and *nextM n* generates a random number in the range $0..n-1$. Note that default definitions are given for each of these functions so that an instance of the class *PRNG* need only supply an implementation for either *nextB* or *nextI*. It should also be noted that each of these functions takes a *PRNG* parameter and returns an updated *PRNG* value as part of its result. It is extremely important that a *PRNG* value should only be used once to generate a random value during a computation.

To support monadic programming, we define the type *SecureRandom* that *wraps* a *PRNG* type and defines it to be an instance of the class *Monad* (Figure 15). This definition is just an adaptation of the standard approach for representing *state transformers* [30] in *Haskell*.

A BBS pseudo-random bit generator can be implemented as an algebraic datatype *BBS* and this data type can be made an instance of the class *PRNG* (Figure 14). The seed information *seed* used by BBS must be relatively prime to n and therefore, *seedBBSRand* will increment *seed* until a suitable value is obtained.

Using *PRNG* and *SecureRandom*, we implement a function *mkPrime* that can be used to generate a random prime (Figure 16). The expression *mkPrime*

[9] This is not a serious problem as suitable functions which offer the required functionality could easily be implemented using the standard class *Random*.

$p\ f\ s$ is used to generate an s-bit prime using the primality test p and satisfying the filter[10] f. The types for primality tests and filters are also given in Figure 16.

To complete the code necessary for key generation, we define the function *mkRSAKey* (Figure 17) such that *mkRSAKey s* will generate an RSA key-pair with an s-bit modulus. This function always uses a public exponent with the value 3 and therefore, requires primes p and q such that $(p-1)$ **mod** $3 = 0$ and $(q-1)$ **mod** $3 = 0$. This ensures that $gcd(3, \phi(pq)) = 1$ as required by RSA.

Complete code and examples are given in [1].

3.4.1 Monadic Programming

For readers not familiar with *monadic programming*, the code presented in section 3.4 looks unnecessarily complex and possibly incorrect. For example, one would normally expect *mkPrime* to take a pseudo-random number generator as a parameter and return an integer value. Instead, however, *mkPrime* returns a *SecureRandom* value which is itself a function that takes a *PRNG* value and returns both a prime and a new *PRNG* value.

This rather complex arrangement allows the monadic style of programming to be used with functions (e.g., *mkPrime* and *mkRSAKey*) that need to keep a *state* consisting of a pseudo-random number generator that is updated each time it is used[11]. For example, the body of the function *mkRSAKey* is implemented using the *Haskell* primitives **do**, **return** and $\leftarrow$ to produce code that is analogous to similar imperative code.

A full explanation of monadic programming is beyond the scope of this paper, but interested readers are referred to papers by Wadler [30, 31].

3.5 Digital Signatures

While RSA can be used to keep information confidential, it is more commonly used to implement digital signatures. To create a digital signature, a user must

[10] When generating primes, it is common to require primes that satisfy specific properties, e.g., the primes used with BBS need to be congruent 3 modulo 4. These requirements can be captured using a suitable filter function.

[11] Since *Haskell* is a pure functional programming language with no imperative features, the idea of a function maintaining a state is purely conceptual.

```
class PRNG g where
   nextB       :: g → (Integer,g)
   nextI       :: Int → g → (Integer,g)
   nextM       :: Integer → g → (Integer,g)
```

Figure 13: The `PRNG` class

```
data BBSRand = BBS Integer Integer -- (modulus,x)

seedBBSRand :: Integer → Integer → BBSRand
seedBBSRand modulus seed  =
             let
               (g,_,_) = gcde seed modulus
             in
                if   g /= 1
                then seedBBSRand modulus (seed + 1)
                else BBS modulus ((seed*seed) `mod` modulus)

nextBBSBit :: BBSRand → (Integer,BBSRand)
nextBBSBit (BBS modulus x) = (x `mod` 2,BBS modulus ((x*x) `mod` modulus))

instance PRNG BBSRand where
  nextB   = nextBBSBit
```

Figure 14: The `BBS` pseudo-random bit generator

encrypt[12] the message m to be signed with their private-key K_{priv}. This produces the signature $\mathcal{D}(K_{priv})(m)$. Since $\mathcal{E}(K_{pub})(\mathcal{D}(K_{priv})(m)) = m$, anyone with access to the user's public-key K_{pub} can decrypt this signature and recover m. Of course, this only works if the message m contains sufficient redundancy, i.e., it must be impossible for an arbitrary piece of ciphertext to be mistaken for a valid signature. Since the original message m can be recovered from the signature, this approach is an example of a *Digital Signature Scheme with Recovery* [19].

RSA can only be used with messages m such that $0 \leq m < n$ where n is

[12]RSA encryption with a private-key is the same as decryption.

```
data  PRNG g ⇒ SecureRandom g a = SecureRandom (g → (a,g))

thenRandom :: PRNG g ⇒ (SecureRandom g a) →
                          (a → (SecureRandom g b)) → (SecureRandom g b)
thenRandom (SecureRandom r) f =
             (SecureRandom (\g → let (v,g') = r g
                                     (SecureRandom r') = f v
                                 in r' g'))

instance PRNG g ⇒ Monad (SecureRandom g) where
  (>>=)    = thenRandom
  return a = (SecureRandom (\g → (a,g)))
```

Figure 15: The `SecureRandom` type

the modulus of the RSA key, so the above approach cannot be applied to large messages. However, if a suitable hash function is used, it is possible to produce a signature by hashing the message m and encrypting the digest. Given an arbitrarily large message m and hash function $\mathcal{H}$, the signature for m is given by $\mathcal{D}(K_{priv})(\mathcal{H}(m))$. To validate a signature, a user with access to the public-key K_{pub} recovers the digest of m from the signature using $\mathcal{E}(K_{pub})(\mathcal{D}(K_{pub})(\mathcal{H}(m)))$ and compares this with the digest that is computed by hashing m directly. If these digests are the same then the signature is valid. Since signature validation requires access to both the signature and the original data, this approach is an example of a *Digital Signature Scheme with Appendix* [19].

With this scheme, an arbitrary ciphertext c can be decrypted using a public-key to produce a digest d. However, for a secure hash function $\mathcal{H}$, it is computationally infeasible to find a message m such that $\mathcal{H}(m) = d$, i.e., there is no need to add redundancy[13] to the digest before it is encrypted. Therefore, we will present a solution in which the digest (represented as an integer value) is encrypted using an RSA private-key.

To implement digital signatures we use an implementation of SHA-1 [7] due

[13]There are a number of other attacks that can be mounted against signatures so it is common to embed the digest in a structure containing the identity of the hash function used to produce the signature. The most widely used format is *PKCS#1* [23].

```
newtype PRNG g ⇒ ProbabilityTest g = PTEST (Integer → (SecureRandom g Bool))

isFermat :: PRNG g ⇒ Int → (ProbabilityTest g)

type PrimeFilter = Integer → Bool

mkPrime :: (PRNG g) ⇒
            (ProbabilityTest g) → PrimeFilter → Int →  (SecureRandom g Integer)

mkPrime (PTEST pTest) filter s =
        let
            try p =
                do
                  b ← pTest p
                  (if b && (filter p)
                   then return p
                   else try (p+2))
        in
         do
            p ← nextOddInteger s
            try p
```

Figure 16: The function `mkPrime`

to Lynagh [18]. This implementation has been slightly modified[14] so that the function *sha1* produces the SHA-1 hash of a string in the form of a large integer value. This gives the functions *sign* that can be used to produce a digital signature and *verify* that can be used to validate a signature (Figure 18).

It should be noted that *sign* and *verify* require the modulus of key-pairs to be greater than the numeric value of the SHA-1 digest. Since SHA-1 produces a 160-bit digest, the modulus needs to be greater than $2^{160}-1$. Since a secure key requires significantly more bits (e.g., $512-1024$) it is extremely unlikely that a randomly generated modulus would be too small. However, the key generation

[14] In addition, some internal functions were modified to prevent stack overflow when hashing large strings.

```
mkRSAKey :: (PRNG g) ⇒ Int → SecureRandom g (RSAPrivateKey,RSAPublicKey)

mkRSAKey s =
     do
       p ← mkPrime (isFermat 20) (\p → (p–1) `mod` 3 /= 0) (s `div` 2)
       q ← mkPrime (isFermat 20) (\p → (p–1) `mod` 3 /= 0) (s `div` 2)
       return (computeRSAKey p q 3)
```

Figure 17: The function `mkRSAKey`

```
sign :: RSAPrivateKey → String → Integer
sign k d = drsa k (sha1 d)

verify :: RSAPublicKey → String → Integer → Bool
verify k d s = (sha1 d) == (ersa k s)
```

Figure 18: The functions `sign` and `verify`

function *mkRSAKey* presented in this paper is not guaranteed to produce a sufficiently large modulus. This property is a result of our definition of an n-bit number. Careful inspection of our code (in particular, the definition of *nextI* in *PRNG*) reveals that an n-bit number is one that can be represented in **at most** n bits, i.e., it is a number in the range $0..2^n - 1$. Our code could easily be modified to remove all possibility of a small modulus being generated.

4 Performance Measurements

The *Haskell* code presented above can be used directly with interactive *Haskell* environments such as *HUGS* [14] or *GHCi* [29]. In particular, students can invoke functions directly with suitable arguments to get a better understanding of how RSA works. In effect, this allows *HUGS* and *GHCi* to be used as *RSA Calculators*.

However, it is also possible to use the *Haskell* code presented above to build applications. As an example, we have produced a small example application that can be used to digitally sign files. This application consists of four programs:

Generate:

That can be used to generate an RSA key-pair which is stored for future use.

Sign:

That can be used to produce the digital signature of a file using the *current* RSA key-pair.

Verify:

That can be used to validate the signature of a file using the *current* RSA key-pair.

Digest:

That can be used to produce the SHA-1 digest of a file.

Details of these programs can be found in [1].

This application has been used to compare the performance of our *Haskell* software to equivalent *Java* software. The full *Haskell* code used to perform these comparisons is presented in [1]. The *Java* code used in the comparisons code is also distributed with this *Haskell* code. The following points should be noted:

1. The *Java* software uses the same algorithms as the *Haskell* software and produces exactly the same output. In particular, random number generators are seeded in the same way and produce the same results.

2. Within this *Java* software, large numbers are represented using the core library class `java.math.BigInteger` and digests are produced using the core library implementation of SHA-1. However, all other software (including RSA encryption and decryption, and the BBS pseudo-random bit generator) are implemented as part of the *Java* code.

The results of these comparisons are presented in Tables 1 and 2. The following points should be noted:

1. Signing and validation in Table 1 uses a 1024-bit key.

2. It is clear from the figures given in Table 1 that the *Haskell* implementation of SHA-1 is considerably slower than the *Java* implementation. However, if

Table 1: Times (millisecs) for hashing, signing and signature validation

	Hashing		*Signing*		*Validation*	
File Size (bytes)	Java	Haskell	Java	Haskell	Java	Haskell
7192	20	200	78	227	20	203
14383	26	400	80	440	29	412
28765	27	820	83	920	42	833

the overhead for hashing in *Haskell* is removed, then we see that encryption with a private-key (signing) and decryption with the public-key (signature validation) take comparable times in both implementations.

3. Decryption (signature validation) is faster than encryption as we use a public exponent of 3.

4. Table 2 contains typical times required to generate key-pairs of different sizes. The *Haskell* implementation is considerably faster than the *Java* implementation as key generation requires a considerable amount of arithmetic on large integer values. In the case of the *Haskell* code compiled with *GHC*, this arithmetic is performed by an arbitrary precision library implemented in *C*. In *Java*, however, the core library class `java.math.BigInteger` is interpreted as bytecode.

The timings presented in tables 1 and 2 are intended to give a rough comparison between the *Haskell* and *Java* programs. In particular, no attempt was made to optimise either the *Haskell* or *Java* code, and no investigation into the performance of the respective runtime environments has been carried out. Therefore, these performance figures should be treated with caution.

5 Conclusions

In this paper we have presented a working implementation of the RSA cryptosystem and a small example application, both written in *Haskell*. This software serves two useful pedagogical purposes:

Table 2: Times (millisecs) for generating keys

Key Size (bits)	Java	Haskell
256	3464	1120
512	14827	5205
768	18312	6433
1024	66975	23707

1. It is a very clear, high-level implementation of RSA that can easily be understood and manipulated by students of cryptography.

2. It is an excellent example of how a functional programming language such as *Haskell* can be used to implement *real* applications.

The *Haskell* code makes use of a wide range of the features of *Haskell*; in particular, the code uses classes and monadic programming.

1. The class construct in *Haskell* is a mechanism that allows function names and operators to be *overloaded*. This is sometimes referred to as *ad-hoc polymorphism* and is different from *general polymorphism* introduced by classes in object-oriented programming languages such as *Java*.

2. *Haskell* is a pure functional programming language with no imperative features. In particular, there is no assignment or concept of state. Monadic programming [30, 31] allows imperative programming involving state transformations to be simulated in *Haskell*.

For students who know or are being taught *Haskell*, the use of these features should not present any problems and in fact, their use is a good example of how such features can be used in real applications. However, for other students their use can be confusing and difficult to understand.

For students of cryptography who have programming experience but not necessarily using functional programming languages, experience suggests that using monadic programming is unnecessarily complex and detracts from the goal of having a simple notation for exploring cryptography. Fortunately, the use of monadic programming is not essential and the *Haskell* code presented above can

easily[15] be rewritten in a more basic functional programming style. Taking this approach it is reasonable to expect students of a cryptography course to learn the subset of *Haskell* needed to implement cryptography.

Of course, implementation exercises in public-key cryptography courses can be implemented in any suitable programming language. *C* [15], *C++* [6] and *Java* [11] (possibly in conjunction with libraries that implement arbitrary precision arithmetic) are commonly used. The advantage of *Haskell* is that the resulting code is much more concise and can be used interactively to explore the behaviour of algorithms.

An alternative approach would be to use mathematical packages such as Mathematica [33] or Maple [32]. However, such packages are proprietary products that use their own, usually complex, notations. In contrast, *Haskell* is a relatively simple notation and is an open standard with a number of freely available implementations. For students with a background in programming, the subset of *Haskell* required is easier to learn than one of these proprietary notations, but for students with experience of one of these packages, it would probably more effective to simply use the package.

The material presented in this paper covers a complete implementation of RSA, however, *Haskell* can also be used to implement other algorithms and techniques used in public-key cryptography. For example, the author has also used *Haskell* for implementations of ElGamal, DSA, elliptic curve cryptography and other signature schemes during a course on public-key cryptography.

Acknowledgements

Many thanks go to the students of ******: Public-Key Cryptography* at

********** for using *Haskell* as their implementation language. Thanks also goes to ***** *****, ***** ******** and a number of anonymous reviewers for commenting on earlier drafts of this paper.

[15]When a more basic style is used, a programmer must explicitly handle the updated "states" of pseudo-random number generators. This can be a source of errors in which pseudo-random number generators are used more than once.

References

[1] D. ****. *Source Code Companion for Implementioning Public-Key Cryptography in Haskell*. **********, **********, ******, *******, 2001.

[2] D. ****. *Source code for an implementation of RSA in* `Haskell`. **********, **********, ******, *******, 2001.
`http://********.*******.***.***/haskell/rsa.html`.

[3] I. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography*. Number 265 in London Mathematical Society Lecture Note Series. Cambridge University Press, 2000.

[4] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, 1976.

[5] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31:469–472, 1985.

[6] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[7] *Federal Information Processing Standards (FIPS) Publication 180-1, Secure Hash Standard (SHA)*, 1995.
`http://www.itl.nist.gov/fipspubs/fip180-1.htm`.

[8] *Federal Information Processing Standards (FIPS) Publication 185, Escrowed Encryption Standard (EES)*, 1994.
`http://www.itl.nist.gov/fipspubs/fip185.htm`.

[9] *Federal Information Processing Standards (FIPS) Publication 186, Digital Signature Standard (DSS)*, 1994.
`http://www.itl.nist.gov/fipspubs/fip186.htm`.

[10] *Federal Information Processing Standards (FIPS) Publication 46-2, Data Encryption Standard(DES)*, 1993.
`http://www.itl.nist.gov/fipspubs/fip46-2.htm`.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. The Java Series. Addison-Wesley, second edition, 2000.

[12] B. Hartley and T. O. Hawkes. *Rings, Modules and Linear Algebra*. Chapman and Hall, 1970.

[13] *ISO/IEC 10118-3:1998 Security Techniques: Hash-functions Part 3: Dedicated hash-functions* , 1998. http://www.esat.kuleuven.ac.be/ bosselae/ripemd160.html.

[14] M. P. Jones and J. C. Peterson. HUGS98*, A functional programming system based on Haskell 98*, 1999. http://www.haskell.org/hugs/.

[15] B. W. Kernighan and D. M. Ritche. *The C Programming Language*. Prentice Hall, 1988.

[16] R. Kumanduri and C. Romero. *Number Theory with Computer Applications*. Prentice Hall, 1998.

[17] X. Lai. On the Design and Security of Block Ciphers. ETH Series in Information Processing 1, 1992.

[18] Ian Lynagh. Oxford University, UK. http://c93.keble.ox.ac.uk/ ian/haskell/.

[19] A.J. Menezes, P. C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. http://cacr.math.uwaterloo.ca/hac/.

[20] S. Peyton Jones et al. *Report on the Programming Language Haskell 98, A Non-strict, Purely Functional Language*, 1999. http://www.haskell.org/onlinereport/.

[21] S. Peyton Jones et al. *Standard Libraries for the Programming Language Haskell 98*, 1999. http://www.haskell.org/onlinereport/.

[22] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[23] RSA Laboratories. *PKCS #1*, 1998. http://www.rsa.com.

[24] B. Schneier. *Applied Cryptography*. John Wiley & Sons, 1996.

[25] J. H. Silverman. *A friendly Introduction to Number Theory*. Prentice Hall, 1997.

[26] D. R. Stinson. *Cryptography Theory and Practice*. The CRC Press Series on Discrete Mathematics and its Applications. CRC Press, 1995.

[27] *Java Cryptography Architecture, API Specification & Reference*, 1999. http://java.sun.com/products/jdk/1.2/docs/guide/security/CryptoSpec.html.

[28] *Java Cryptography Extension 1.2*, 2001. http://java.sun.com/products/jce/index-12.html.

[29] The GHC team. *The Glasgow Haskell Compiler User's Guide*, 1999. http://www.haskell.org/ghc/.

[30] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.

[31] Philip Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.

[32] Waterloo Maple Inc. *Waterloo Maple*. http://www.maplesoft.com/.

[33] Wolfram Research Inc. *Mathematica*. http://www.wolfram.com/.