International Journal on New Computer Architectures and Their Applications (IJNCAA) 1(1): 34-51
The Society of Digital Information and Wireless Communications, 2011 (ISSN 2220-9085)

DNA Security using Symmetric and Asymmetric Cryptography

Radu Terec[1], Mircea-Florin Vaida[1], Lenuta Alboaie[2], Ligia Chiorean[1]
[1]Technical University of Cluj-Napoca, Faculty of Electronics, Telecommunications and Information Technology, Departament of Communications, 26 – 28 Gh. Baritiu, 400027, Cluj-Napoca, Romania, Phone: (+40) 264 401810,
Mircea.Vaida@com.utcluj.ro (corresponding author), RaduTerec@gmail.com, Chiorean.Ligia@com.utcluj.ro
[2]Alexandru Ioan Cuza University of Iasi, Romania, Faculty of Computer Science, Berthelot, 16, Iasi, Romania, adria@info.uaic.ro

ABSTRACT

This paper presents alternative security methods based on DNA. From the available alternative security methods, symmetric DNA algorithms were developed and implemented. The first symmetric DNA algorithm was implemented in the Java language, while the second DNA algorithm was implemented in BioJava and MatLab. Comparisons have been made between the performances of different standard symmetrical algorithms and the DNA proposed algorithms. As a new step to enhance the security, an asymmetric key generation inside a DNA security algorithm is presented. The asymmetric key generation algorithm starts from a password phrase. The asymmetric DNA algorithm proposes a mechanism which makes use of more encryption technologies. Therefore, it is more reliable and more powerful than the OTP DNA symmetric algorithms.

KEYWORDS

DNA security, symmetric cryptography, OTP, asymmetric cryptography, BioJava

1 INTRODUCTION

The security of a system is essential nowadays. With the growth of the information technology (IT) power, and with the emergence of new technologies, the number of threats a user is supposed to deal with grew exponentially. It doesn't matter if we talk about bank accounts, social security numbers or a simple telephone call. It is important that the information is known only by the intended persons, usually the sender and the receiver.

A security system may have a lot of weak spots: the place where the ciphers are stored, the random number generator, the strength of the used algorithms and so on. The job of the security designer is to make sure none of these weaknesses gets exploited.

Based on the confidentiality property in the domain of security the symmetrical and asymmetrical cryptographic algorithms are used. Cryptography consists in processing plain information [1], [2], applying a cipher and producing encoded output, meaningless to a third-party who does not know the key.

In cryptography both encryption and decryption phase are determined by one or more keys. Depending on the type of keys used, cryptographic systems may be classified in:

a) Symmetric systems

-use the same key to encrypt and decrypt data

-symmetric key encryption algorithms (also called ciphers) process plain text

with the secret key to create encrypted data called *ciphertext*
-are extremely fast and well suited for encrypting large quantities of data
-They are vulnerable when transmitting the key
-examples: DES, RC2, 3DES
-PBE (password based encryption) algorithms are derived from symmetric algorithms; such algorithms use a salt (random bytes) and a number of iterations to generate a key

b) Asymmetric systems
-overcome symmetric encryption's most significant disability: the transmission of the symmetric key
-rely on key pairs (contains a public and a private key)
-the public key can be freely shared because it cannot be easily abused, even by an attacker
-messages encrypted with the public key can be decrypted only with the private key
-so, anyone can send encrypted messages, but they can be decrypted by only 1 person
-are not as fast, but are much more difficult to break
-common use: encrypt and transfer a symmetric key (used by HTTPS and SSL).

Symmetrical algorithms use the same key to encrypt and decrypt the data, while asymmetric algorithms use a public key to encrypt the data and a private key to decrypt it. By keeping the private key safe, you can assure that the data remains safe, [3]. The disadvantage of asymmetric algorithms is that they are computationally intensive. Therefore, in security a combination of asymmetric and symmetric algorithms is used.

Another way of ensuring the security of a system is to use a *digital signature*. The signature is applied to the whole document, so if the signature is altered, the document becomes unreadable.

In the future it is most likely that the computer architecture and power will evolve. Such systems might drastically reduce the time needed to compute a cryptographic key. As a result, security systems need to find new techniques to transmit the data securely without relying on the existing pure mathematical methods, [4].

We therefore use alternative security concepts [5]. The major algorithms which are accepted as alternative security are the elliptic, vocal, quantum and DNA encryption algorithms. Elliptic algorithms are used for portable devices which have a limited processing power, use a simple algebra and relatively small ciphers.

The quantum cryptography is not a quantum encryption algorithm but rather a method of creating and distributing private keys. It is based on the fact that photons send towards a receiver changing irreversibly their state if they are intercepted. Quantum cryptography was developed starting with the 70s in Universities from Geneva, Baltimore and Los Alamos.

In [6] two protocols are described, BB84 and BB92, that, instead of using general encryption and decryption techniques, verify if the key was intercepted. This is possible because once a photon is duplicated, the others are immediately noticed. However, these techniques are still vulnerable to the Man-in-the-Middle and DoS attack.

DNA Cryptography is a new field based on the researches in DNA computation [7] and new technologies like: PCR (Polymerase Chain Reaction), Microarray, etc. DNA computing has a high level computational ability and is capable of storing huge amounts of data. A gram of DNA contains 10^{21} DNA bases, equivalent to 10^{8} terabytes of data. In DNA cryptography we use existing biological information from DNA public databases to encode the plaintext [8], [9].

The cryptographic process can make use of different methods, [10]. In [5] the *one-time pads* (OTP) algorithms are described, which is one of the most efficient security algorithms, while in [11] a method based on the DNA splicing technique is detailed. In the case of the *one-time pad* algorithms, the plaintext is combined with a secret random key or *pad* which is used only once. The pad is combined with the plaintext using a typical modular addition, or an XOR operation, or another technique. In the case of [11] the start codes and the pattern codes specify the position of the introns, so they are no longer easy to find. However, to transmit the spliced key, they make use of public-key secured channel.

Additionally, we will describe an algorithm which makes use of asymmetric cryptographic principles. The main idea is to avoid the usage of both purely mathematical symmetric and asymmetric algorithms and to use an advanced asymmetric algorithm based on DNA. The speed of the algorithm should be quite high because we make use of the powerful parallel computing possibilities of the DNA. Also, the original asymmetric keys are generated starting from a user password to avoid their storage.

The paper is structured in 5 sections. In section 2 we present some general aspects about the used technologies: Java security API, genetic code and BioJava. In section 3 we present a Java algorithm implementation based on a DNA mechanism, and another algorithm for the symmetric DNA cryptography, using a BioJava implementation (similar in MatLab implementation). We will also expose the limitation imposed by these platforms. In section 4 we describe an advanced asymmetric DNA encryption algorithm. We will conclude this paper in section 5 where a comparison between the obtained results is made and the conclusions and possible continuations of our work are presented.

2 USED TECHNOLOGIES

2.1. The Java Cryptography Architecture

The Security API (Application Programming Interface) is a core API of the Java programming language, built around the *java.security* package, [12]. This API is designed to allow developers to incorporate both low-level and high-level security functionality into their programs.

The Java Cryptography Extension (JCE) extends the JCA API to include APIs for encryption, key exchange, and Message Authentication Code (MAC). Together, the JCE and the cryptography aspects of the SDK provide a complete, platform-independent cryptography API. JCE was previously an optional package (extension) to the Java 2 SDK, Standard Edition, versions 1.2.x and 1.3.x. JCE has been integrated into the Java 2 SDK, v 1.4.

The Java Cryptography Architecture (JCA) was designed around these principles: implementation independence and interoperability; algorithm independence and extensibility. Implementation independence and algorithm independence are complementary; when complete algorithm-independence is not possible, the JCA provides standardized, algorithm-specific APIs. When implementation-independence is not desirable, the JCA lets developers indicate a specific implementation.

The Java Cryptography Architecture introduced the notion of a *Cryptographic Service Provider*, or simply *provider*. This term refers to a package (or a set of packages) that supply a concrete implementation of a subset of the cryptography aspects of the Security API. It has methods for accessing the provider name, version number, and other information.

For each engine class in the API (an engine class provides the interface to the functionality of a specific type of cryptographic service, independent of a particular cryptographic algorithm), a particular implementation is requested and instantiated by calling a *getInstance()* method on the engine class, specifying the name of the desired algorithm and, optionally, the name of the provider (or the `Provider` class) whose implementation is desired.

If no provider is specified, *getInstance()* searches the registered providers for an implementation of the requested cryptographic service associated with the named algorithm. In any given Java Virtual Machine (JVM), providers are installed in a given *preference order*, the order in which the provider list is searched if a specific provider is not requested. For example, suppose there are two providers installed in a JVM, PROVIDER_1 and PROVIDER_2.

From the core classes specified by the JCA a special attention will be drawn to the following classes: The Security, KeyGenerator and the Cipher class.

a) The Security Class

The *Security* class manages installed providers and security-wide properties. It only contains static methods and is never instantiated. The methods for adding or removing providers, and for setting *Security* properties, can only be executed by a trusted program. Currently, a "trusted program" is either a local application not running under a security manager, or an applet or application with permission to execute the specified method (see below).

The determination that code is considered trusted to perform an attempted action (such as adding a provider) requires that the applet is granted permission for that particular action.

b) The KeyGenerator Class

This class is used to generate secret keys for symmetric algorithms, necessary to encrypt the plaintext. KeyGenerator objects are created using the *getInstance()* factory method of the KeyGenerator class. *getInstance()* takes as its argument the name of the symmetric algorithm for which the key was generated. Optionally, a package provider name may be specified:

```
public static KeyGenerator
getInstance(String alg, String
provider);
```

There are two ways to generate a key: algorithm-independent manner and algorithm-specific manner. In the algorithm-independent manner, all key generators share the concepts of a key size and a source of randomness.

In the algorithm-specific manner, for situations where a set of algorithm-specific parameters already exists, there are two *init* methods that have an *AlgorithmParameterSpec* argument:

```
public void init
(AlgorithmParameterSpec params);

public void init
(AlgorithmParameterSpec params,
SecureRandom random);
```

In case the client does not explicitly initialize the KeyGenerator each provider must supply a default initialization

Wrapping a key enables secure transfer of the key from one place to another, fact which can be used if one needs to send data over a media available to more people like a network, for example.

c) The Cipher Class

This class forms the core of the JCE framework. It establishes a link between the data, the algorithm and the key used whether it is for encoding, decoding or wrapping. Cipher objects are created using the *getInstance()* factory method of the Cipher class.

```
public static Cipher
getInstance(String
transformation,String provider);
```

A Cipher object obtained via *getInstance()* must be initialized for one of four modes: ENCRYPT_MODE = Encryption of data, DECRYPT_MODE = Decryption of data, WRAP_MODE = Wrapping a Key into bytes so that the key can be securely transported, UNWRAP_MODE = Unwrapping of a previously wrapped key into a *java.security* Key object.

2.2. General Aspects about Genetic Code

There are 4 nitrogenous bases used in making a strand of DNA. These are adenine (A), thymine (T), cytosine (C) and guanine (G). These 4 bases (A, T, C and G) are used in a similar way to the letters of an alphabet. The sequence of these DNA bases will code specific genetic information [8].

In our previous work we used a one-time pad, symmetric key cryptosystem [13], [14], [15]. In the OTP algorithm, each key is used just once, hence the name of OTP. The encryption process uses a large non-repeating set of truly random key letters. Each pad is used exactly once, for exactly one message. The sender encrypts the message and then destroys the used pad. As it is a symmetric key cryptosystem, the receiver has an identical pad and uses it for decryption. The receiver destroys the corresponding pad after decrypting the message. New message means new key letters. A cipher text message is equally likely to correspond to any possible plaintext message. Cryptosystems which use a secret random OTP are known to be perfectly secure.

By using DNA with common symmetric key cryptography, we can use the inherent massively-parallel computing properties and storage capacity of DNA, in order to perform the encryption and decryption using OTP keys. The resulting encryption algorithm which uses DNA medium is much more complex than the one used by conventional encryption methods.

To implement and exemplify the OTP algorithm, we downloaded a chromosome from the open source NCBI GenBank, [16]. As stated, in this algorithm the chromosomes are used as cryptographic keys. They have a small dimension and a huge storage capability. There is a whole set of chromosomes, from different organisms which can be used to create a unique set of cryptographic keys. In order to splice the genome, we must know the order in which the bases are placed in the DNA string.

The chosen chromosome was "*Homo sapiens FOSMID clone ABC24-1954N7 from chromosome 1*". Its length is high enough for our purposes (37983 bases).

GenBank offers different formats in which the chromosomal sequences can be downloaded:

- GenBank,
- GenBank Full,
- FASTA,
- ASN.1.

We chose the FASTA format because it's easier to handle and manipulate. To manipulate the chromosomal sequences we used BioJava API methods, a framework for processing DNA sequences. Another API which can be used for managing DNA sequences is offered by MatLab. Using this API, a dedicated application has been implemented [13].

In MatLab, the plaintext message was first transformed in a bit array. An encryption unit was transformed into an 8 bit length ASCII code. After that, using functions from the Bioinformatics Toolbox, each message was transformed from binary to DNA alphabet. Each character was converted to a 4-letter DNA sequence and then searched in the chromosomal sequence used as OTP, [14].

2.3. BioJava API

The core of BioJava is actually a symbolic alphabet API, [17]. Here, sequences are represented as a list of references to singleton symbol objects that are derived from an alphabet. The symbol list is stored as often as possible. The list is compressed and uses up to four symbols per byte.

Besides the fundamental symbols of the alphabet (A, C, G and T as mentioned earlier), the BioJava alphabets also contain extra symbol objects which represent all possible combinations of the four fundamental symbols. The structure of the BioJava architecture together with its most important APIs is presented below:

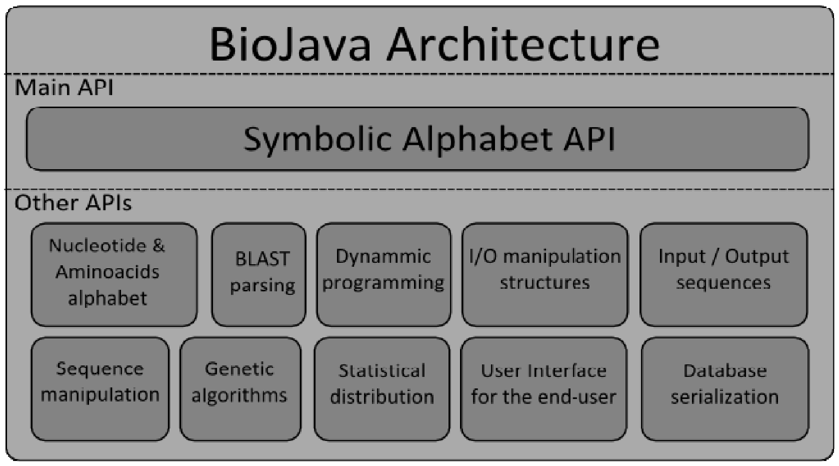


Figure 1. The BioJava Architecture

By using the symbol approach, we can create higher order alphabets and symbols. This is achieved by multiplying existing alphabets. In this way, a codon can be treated as nothing more than just a higher level alphabet, which is very convenient in our case. With this alphabet, one can create views over sequences without modifying the underlying sequence.

In BioJava a typical program starts by using the sequence input/output API and the sequence/feature object model. These mechanisms allow the sequences to be loaded from a various number of file formats, among which is FASTA, the one we used. The obtained results can be once more saved or converted into a different format.

3 DNA CRYPTOGRAPHY IMPLEMENTATIONS

In this chapter we will start by presenting the initial Java implementation of the symmetric OTP encryption algorithm, [18]. We will then continue by describing the corresponding BioJava (and Matlab) implementation and some drawbacks of this symmetric algorithm.

3.1. Java Implementation

One approach of the DNA Cryptography is a DNA-based symmetric cryptographic algorithm. This algorithm involves three steps: key generation, encryption and decryption. In fact, the encryption process makes use of two classic cryptographic algorithms: the one-time pad, and the substitution cipher.

Due to the restrictions that limit the use of JCE, [19], the algorithm was developed using OpenJDK, which is based on the JDK 7.0 version of the Java platform and does not enforce certificate verification.

In order to generate random data we use the class *SecureRandom* housed in the *java.security* package, class which is designed to generate cryptographically secure random numbers. The next step is translating this key in DNA language by limiting the range of numbers to [0, 3] and associating a letter to each number as following:

Table 1. Translation table

Number	Corresponding letter
0	a
1	c
2	g
3	t

At this time is very important to know that the length of the key must be exactly the same as the length of the plaintext. In this case, the plaintext is the secret message, translated according to the substitution alphabet.

Therefore, the length of the key is three times the length of the secret message. The user may choose the length of the key, the only restriction being that this must be a multiple of three.

Because the key must have three times the length of the messages, when trying to send very long messages, the length of the key would be huge. For this reason, the message is broken into fixed-size blocks of data. The cipher encrypts or decrypts one block at a time, using a key that has the same length as the block.

The implementation of block ciphers raises an interesting problem: the message we wish to encrypt will not always be a multiple of the block size. To compensate for the last incomplete block, padding is needed. A padding scheme specifies exactly how the last block of plaintext is filled with data before it is encrypted. A corresponding procedure on the decryption side removes the padding and restores the plaintext's original length. However, this DNA Cipher will not use a padding scheme but a shorter version (a fraction) of the original key. The only mode of operation implemented by the DNA Cipher is ECB (Electronic Code Book). This is the simplest mode, in which each block of plaintext encrypts to a block of ciphertext. ECB mode has the disadvantage that the same plaintext will always encrypt to the same ciphertext, when using the same key.

As we mentioned, the DNA Cipher applies a double encryption in order to secure the message we want to keep secret. The first encryption step uses a substitution cipher.

For applying the substitution cipher it was used a HashMap Object. HashMap is a *java.util* class that implements the Map interface. These objects associate a specified value to a specified unique key in the map.

One possible approach is representing each character of the secret message by a combination of 3 bases on DNA, as shown in the table below:

Table 2. The substitution alphabet

a - cga	l - tgc	w - ccg	3 - gac
b - cca	m -tcc	x - cta	4 - gag
c - gtt	n - tct	y - aaa	5 - aga
d - ttg	o -gga	z - ctt	6 - tta
e -ggc	p -gtg	_ - ata	7 - aca
f - ggt	q -aac	, - tcg	8 - agg
g - ttt	r - tca	. - gat	9 - gcg
h -cgc	s -acg	: - gct	space- ccc
i - atg	t - ttc	0 - act	
j - agt	u - ctg	1 - acc	
k -aag	v - cct	2 - tag	

Given the fact that this cipher replaces only lowercase characters with their corresponding triplet and that in most messages we encounter also upper case letters, the algorithm first transforms all the letters of the given secret message into lowercase letters.

The result after applying the substitution cipher is a string containing characters from the DNA alphabet (a, c, g, t). This will further be transformed into a byte array, together with the key. The exclusive or operation (XOR) is then applied to the key and the message in order to produce the encrypted message.

When decrypting an encrypted message, it is essential to have the key and the substitution alphabet. While the substitution alphabet is known, being public, the key is kept secret and is given only to the addressee. Any malicious third party won't be able to decrypt the message without the original key.

The received message is XOR-ed with the secret key resulting a text in DNA alphabet. This text is then broken into groups of three characters and with the help of the reverse map each group will be replaced with the corresponding letter. The reverse map is the inverse of the one used for translating the original message into a DNA message. This way the receiver is able to read the original secret message.

3.2 BioJava Implementation

In this approach [20], we use more steps to obtain the DNA code starting from the plaintext. For each character from the message we wish to encode, we first apply the *get_bytes()* method which returns an 8bit ASCII string of the character we wish to encode. Further, we apply the *get_DNA_code()* method which converts the obtained 8 bit string, corresponding to an ASCII character, into DNA alphabet. The function returns a string which contains the DNA-encoded message.

The *get_DNA_code()* method is the main method for converting the plaintext to DNA encoded text. For each 2 bits from the initial 8 bit sequence, corresponding to an ASCII character, a specific DNA character is assigned: 00 – A, 01 – C, 10 – G and 11 – T. Based on this process we obtain a raw DNA message.

Table 3. DNA encryption test sequence

```
Plaintext message: „test"
ASCII message: 116 101 115
116
Raw DNA message:
„TCACGCCCTATCTCA"
```

The coded characters are searched in the chromosome chosen as session key at the beginning of the communication. The raw DNA message is split into groups of 4 bases. When such a group is found in the chromosome, its base index is stored in a vector. The search is made between the first characters of the chromosome up to the 37983th. At each new iteration, a 4 base segment is compared with the corresponding 4 base segment from the raw DNA message. So, each character from the original string will have an index vector associated, where the chromosome locations of that character are found.

The *get_index()* method effectuates the parsing – the comparison of the chromosomal sequences and creates for each character an index vector. To parse the sequences in the FASTA format specific BioJava API methods were used.

BioJava offers us the possibility of reading the FASTA sequences by using a FASTA stream which is obtained with the help of the SeqIOTools class. We can pass through each of the sequences by using a *SequenceIterator* object. These sequences are then loaded into an *Sequence* list of objects, from where they can be accessed using the *SequneceAt()* mrthod.

In the last phase of the encryption, for each character of the message, a random index from the vector index is chosen. We use the *get_random()* method for this purpose. In this way, even if we would use the same key to encrypt a message, we would obtain a different result because of the random indexes.

Since the algorithm is a symmetric one, for the decryption we use the same key as for encryption. Each index received from the encoded message is actually pointing to a 4 base sequence, which is the equivalent of an ASCII character.

So, the *decode()* method realizes following operations: It will first extract the DNA 4 base sequences from the received indexes. Then, it will convert the obtained raw DNA message into the equivalent ASCII-coded message. From the ASCII coded message we finally obtain the original plaintext. And with this, the decryption step is completed.

The main vulnerability of this algorithm is that, if the attacker intercepts the message, he can decode the message himself if he knows the coding chromosomal sequence used as session key.

4 BIOJAVA ASYMMETRIC ALGORITHM DESCRIPTION

In this chapter we will present in detail an advanced method of obtaining DNA-encoded messages. It relies on the use of an asymmetric algorithm and on key generation starting from a user password.

We will also present a pseudo-code description of the algorithm.

4.1 Asymmetric Key Generation

Our first concern when it comes to asymmetric key algorithms was to develop a way in which the user was no longer supposed to deal with key management authorities or with the safe storage of keys. The reason behind this decision is fairly simple: both methods can be attacked. Fake authorities can pretend to be real key-management authorities and intruders may breach the key storage security. By intruders we mean both persons who have access to the computer and hackers, which illegally accessed the computer.

To address this problem, we designed an asymmetric key generation algorithm starting from a password. The method has some similarities with the RFC2898 symmetric key derivation algorithm [21]. The key derivation algorithm is based on a combination of hashes and the RSA algorithm. Below we present the basic steps of this algorithm:

- **Step 1:** First, the password string is converted to a byte array, hashed using SHA256 and then transformed to BigInteger number. This number is transformed in an odd number, *tmp*, which is further used to apply the RSA algorithm for key generation.
- **Step 2**: Starting from *tmp* we search for 2 random pseudo-prime number *p* and *q*. The relation between *tmp, p* and *q* is simple: $p < tmp < q$. To spare the computational power of the device, we do not compute traditionally if *p* and *q* are prime but make primality tests.
- A primality test determines the probability according to which a number is prime. The sequence of the primality test is the following: First, trial divisions are carried out using prime numbers below 2000. If any of the primes divides this BigInteger, then it is not prime. Second, we perform base 2 strong pseudo-prime test. If this BigInteger is a base 2 strong pseudo-prime, we proceed on to the next step. Last, we perform the strong Lucas pseudo-prime test. If everything goes well, it returns true and we declare the number as being pseudo-prime.
- **Step 3**: Next, we determine Euler totient: $phi = (p - 1) * (q - 1)$; and $n = p*q$;
- **Step 4**: Next, we determine the public exponent, *e*. The condition imposed to *e* is to be coprime with *phi*.
- **Step 5**: Next, we compute the private exponential, *d* and the CRT (Chinese Reminder Theorem) factors: *dp*, *dq* and *qInv*.
- **Step 6**: Finally, all computed values are written to a suitable structure, waiting further processing.
- The public key is released as the public exponent, *e* together with *n*.
- The private key is released as the private exponent, *d* together with *n* and the CRT factors.

The scheme of this algorithm is presented below:

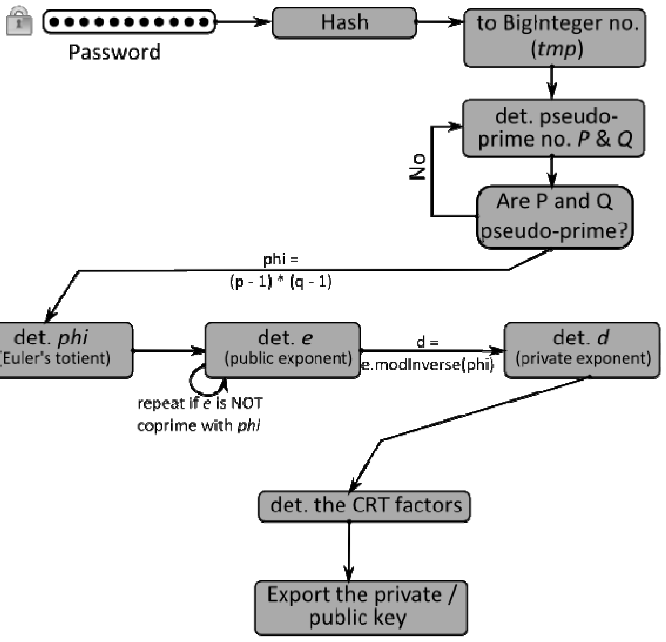

Password
Hash
to BigInteger no. (tmp)
det. pseudo-prime no. P & Q
No
Are P and Q pseudo-prime?
phi = (p - 1) * (q - 1)
det. phi (Euler's totient)
det. e (public exponent)
repeat if e is NOT coprime with phi
d = e.modInverse(phi)
det. d (private exponent)
det. the CRT factors
Export the private / public key

Figure 2. Asymmetric RSA compatible key generation

In comparison with the RFC2898 implementation, here we no longer use several iterations to derive the key. This process has been shown to be time consuming and provide only little extra security. We therefore considered it safe to disregard it.

The strength of the key-generator algorithm is given by the large pseudo-prime numbers it is using and of course, by the asymmetric algorithm. By using primality tests one can determine with a precision of 97 – 99% that a number is prime. But most importantly, the primality tests save time. So, the average computation time, including appropriate key export, for the whole algorithm is 143 ms. After the generation process was completed, the public or private key can be retrieved using the static *ToXmlString* method.

Next, we will illustrate the algorithm through a short example. Suppose the user password is "DNACryptography". Starting from this password, we compute its hash with SHA256. The result is shown below. This hashed password is converted into the BigInteger number *tmp*. Starting from it, and according to the algorithm described above, we generate the public exponent *e* and the private exponent *d*.

Table 4. Asymmetric DNA encryption test sequence

```
user password: “DNACryptography”
hashed password:
“ed38f5aa72c3843883c26c701dfce03
e0d5d6a8d”
tmp =
845979413928639845587469165925 71
6582498797231629929694

4675620251788137567635972662082 9
8952112229
e = 1063
d =
622097271837183006931454033440 94
0850476686457179854307 8

206793184864616193003378707252 34
7966098729919152520454 2

432742920262247220738768537831 77
3689099825753872069076 5
4661581238681185724277829 35
```

We conducted several tests and the generated keys match the PKCS #5 specifications. Objects could be instantiated with the generated keys and used with the normal system-build RSA algorithm.

4.2 Asymmetric DNA Algorithm

The asymmetric DNA algorithm proposes a mechanism which makes use of three encryption technologies. In short, at the program initialization, both the initiator and its partner generate a pair of asymmetric keys. Further, the initiator and its partner negotiate which symmetric algorithms to use, its specifications and of course, the codon sequence where the indexes of the DNA bases will be looked up. After this initial negotiation is completed, the communication continues with normal message transfer. The normal message transfer supposes that the data is symmetrically encoded, and that the key with which the data was encoded is asymmetrically encoded and attached to the data. This approach was first presented in [22].

Next, we will describe the algorithm in more detail and also provide a pseudo-code description for a better understanding.

Step 1: At the startup of the program, the user is asked to provide a password phrase. The password phrase can be as long or as complicated as the user sees fit. The password phrase will be further hashed with SHA256.

Step 2: According to the algorithm described in section 4.1, the public and private asymmetric keys will be generated. Since the pseudo-prime numbers p and q are randomly chosen, even if the user provides the same password for more sessions, the asymmetric keys will be different.

Step 3: The initiator selects which symmetric algorithms will be used in the case of normal message transfer. He can choose between 3DES, AES and IDEA. Further, he selects the time after which the symmetric keys will be renewed and the symmetric key length. Next, he will choose the codon sequence where the indexes will be searched. For all this options appropriate visual selection tools are provided.

Step 4: The negotiation phase begins. The initiator sends to its partner its public key. The partner responds by encrypting his own public key with the initiators public key. After the initiator receives the partner's public key, he will encrypt with it the chosen parameters. Upon receiving the parameters of the algorithms, the partner may accept or propose his own parameters. In case the initiators parameters are rejected, the parties will chose the parameters which provide the maximum available security.

Step 5: The negotiation phase is completed with the sending of a test message which is encrypted like any regular message would be encrypted. If the test message is not received correctly by any of the two parties or if the message transfer takes too much time, the negotiation phase is restarted. In this way, we protect the messages from tampering and interception.

Step 6: The transmission of a normal message. In this case, the actual data will be symmetrically encoded, according to the specifications negotiated before. The symmetric key is randomly generated at a time interval t. The symmetric key is encrypted with the partner's public key and then attached to the message. So, the message consists in the data, encrypted with a symmetric key and the symmetric key itself, encrypted with the partner's public key. We chose to adopt this mechanism because symmetric algorithms are faster than asymmetric ones. Still, in this scenario, the strength of the algorithm is equivalent to a fully asymmetric one because the symmetric key is encrypted asymmetrically. The procedure is illustrated below:

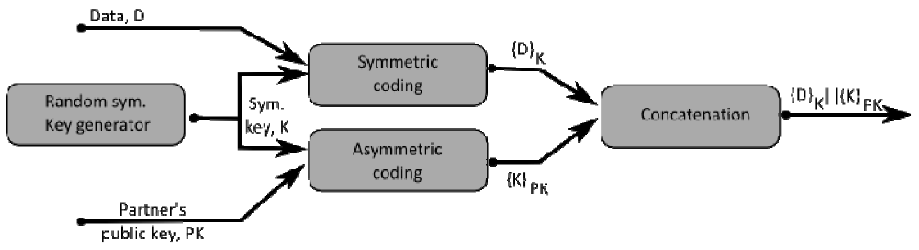


Figure 3. Encryption scheme

Next, the obtained key will be converted into a byte array. The obtained array will be converted to a raw DNA message, by using a substitution alphabet. Finally, the raw DNA message is converted to a string of indexes and then transmitted.

The decryption process is fairly similar. The user converts the index array back to raw DNA array and extracts the ASCII data. From this data he will decipher the symmetric key used for that encryption, by using its private key. Finally the user will obtain the data by using the retrieved symmetric key. At the end of the communication, all negotiated data is disregarded (symmetric keys used, the asymmetric key pair and the codon sequence used).

5 CONCLUSIONS AND COMPARED RESULTS

In this chapter we will present the results we obtained for the symmetric algorithm implementation along with the conclusions of our present work.

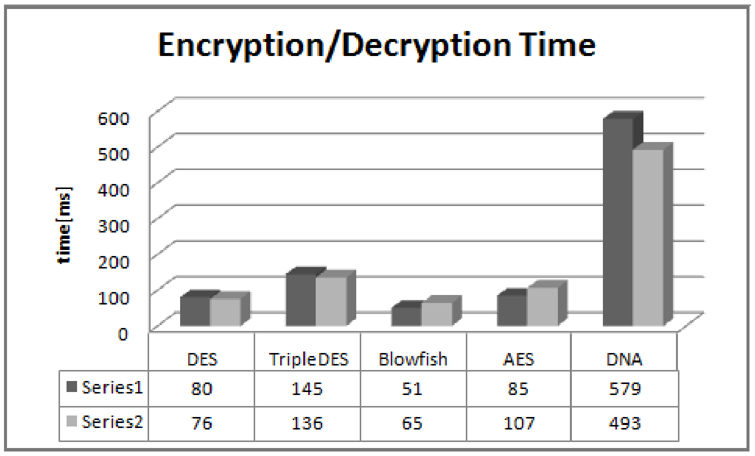
The symmetric OTP DNA algorithm based on Java Cryptography Architecture was first tested, [14]. The purpose is to compare the time required to complete the encryption/ decryption in the case of the DNA Cipher with the time required by other classical encryption algorithms.

The secret message used with all five ciphers was:

„TAACAGATTGATGATGCATG
AAATGGGCCCATGAGTGGCTCCT
AAAGCAGCTGCTtACAGATTGATG
ATGCATGAAATGGGgggtggccaggggt
ggggggtgagactgcagagaaaggcagggctggttc
ataacaagctttgtgcgtcccaatatgacagctgaagttt
tccaggggctgatggtgagccagtgagggtaagtaca
cagaacatcctagagaaaccctcattccttaaagattaa
aaataaagacttgctgtctgtaagggattggattatcctat
ttgagaaattctgttatccagaatggcttaccccacaatg
ctgaaaagtgtgtaccgtaatctcaaagcaagctcctcc
tcagacagagaaacaccagccgtcacaggaagcaaa
gaaattggcttcacttttaaggtgaatccagaacccagat
gtcagagctccaagcactttgctctcagctccacGCA
GCTGCTTTAGGAGCCACTCATGaG
".

The tests ran on a system with the following specifications:

Intel Pentium 4 CPU, 3.00 GHz, RAM: 1,5GB, OS: Ubuntu 9.04



	DES	TripleDES	Blowfish	AES	DNA
■ Series1	80	145	51	85	579
■ Series2	76	136	65	107	493

Figure 4. Encryption/Decryption time for DNA and classical ciphers.

As seen on Figure 4, the DNA Cipher requires a longer time for encryption and decryption, comparatively to the other ciphers. We would expect these results because of the platform used for developing this algorithm. JCA contains the classes of the security package Java 2 SDK, including engine classes. The methods in the classes that implement cryptographic services are divided into two groups. The first group is represented by the APIs (Application Programming Interface). It consists of public methods that can be used by the instances of these classes. The second group is represented by the SPIs (Service Provider Interface)- a set of methods that must be implemented by the derived classes. Each SPI class is abstract. In order to implement a specific service, for a specific algorithm, a provider must inherit the corresponding SPI class and implement all the abstract methods. All these methods process array of bytes while the DNA Cipher is about strings. The additional conversions from string to array of bytes and back make this cipher to require more time for encryption and decryption then other classic algorithms.

To emphasize the difference between DNA and classical algorithms a dedicated application (*SmartCipher*) was developed.

The user has the possibility to enter the text in plain format in the first box and then choose a suitable algorithm to encrypt his text. The encrypted text can be visualized in the second box, while in the third one the user can verify if the decryption process was successful.
An interesting feature of the dedicated application is that it shows the encryption and decryption time. Based on this criterion and the strength of the cipher, the user can estimate the efficiency of the used algorithm.

In the second case considering the symmetrical BioJava mechanism, our first goal was to compare the time required to complete the encryption/ decryption process. We compared the execution time of the DNA Symmetric Cipher with the time required by other classical encryption algorithms. We chose a random text of 360 characters, in string format which was applied to all tests.

The testing sequence is:

Table 5. Testing sequence

```
k39pc3xygfv(!x|jl+qo|9~7k9why(kt
r6pkiaw|gwnn&aw+be|r|*4u+rz$
wm)(v_e&$dz|hc7^+p6%54vp*g*)kzlx
!%4n4bvb#%vex~7c^qe_d745h40i
$_2j*6t0h$8o!c~9x4^2srn81x*wn9&k
%*oo_co(*~!bfur7tl4udm!m4t+a
|tb%zho6xmv$6k+#1$&axghrh*_3_zz@
0!05u*|an$)5)k+8qf0fozxxw)_u
pryjj7_|+nd_&x+_jeflua^^peb_+%@0
3+36w)$~j715*r)x(*bumozo#s^j
u)6jji@xa3y35^$+#mbyizt*mdst&h|h
bf6o*)r2qrwm10ur+mbezz(1p7$f
```

To be able to compute the time required for encryption and decryption, we used the public static *nanoTime*() method from the *System* class which gives the current time in nanoseconds. We called this method twice: once before instantiating the Cipher object, and one after the encryption. By subtracting the obtained time intervals, we determine the execution time.

It is important to understand that the execution time varies depending on the used OS, the memory load and on the execution thread management. We therefore measured the execution time on 3 different machines:

- **System 1:** Intel Core 2 Duo 2140, 1.6 GHz, 1 Gb RAM, Vista OS
- **System 2:** Intel Core 2 Duo T6500, 2.1 GHz, 4 Gb RAM, Windows 7 OS
- **System 3:** Intel Dual Core T4300, 2.1 GHz, 3 Gb RAM, Ubuntu 10.04 OS

Next, we present the execution time which was obtained for various symmetric algorithms in the case of the first, second and the third system, for different cases:

Table 6. Results obtained for System 1

Analysis results for Vista OS							
DES	Encryption	50	26	1.03	0.81	0.84	0.84
	Decryption	1.63	0.35	0.33	0.32	0.34	0.36
AES	Encryption	80	26	0.92	0.95	0.88	0.54
	Decryption	27	2.09	0.30	22.26	0	0.14
Blowfish	Encryption	65	10.91	25	24	0.15	1.45
	Decryption	3	1.87	1.72	29	1.09	1
3DES	Encryption	82	24	2.41	25	2.12	1.42
	Decryption	1.56	1.42	26	1.23	1.41	0.66
BIO sym. algorithm	Encryption	4091	4871	4875	4969	4880	4932
	Decryption	6.29	4.19	4.19	4.19	4.19	4.19

Table 7. Results obtained for System 2

Analysis results for Windows 7							
DES	Encryption	34	1.43	1.09	1.2	1.73	1.19
	Decryption	0.75	0.37	0.44	0.42	0.38	0.37
AES	Encryption	28	1.3	1.16	0.07	1.77	0.82
	Decryption	0.12	0.14	2.09	0.9	2.09	0.16
Blowfish	Encryption	22	28.4	6.2	4	1.6	2.83
	Decryption	2.24	2.21	1.8	1.8	1.8	1.71
3DES	Encryption	41	6.59	2.78	2.62	2.69	2.12
	Decryption	1.12	1.78	1.24	1.74	1.48	1
BIO sym. algorithm	Encryption	3970	3884	3887	3901	3900	3910
	Decryption	4.19	4.19	4.19	2.09	4.19	2.09

Table 8. Results obtained for System 3

Analysis results for Ubuntu 10.04							
DES	Encryption	12.64	0.9	0.61	0.59	0.61	0.56
	Decryption	1.24	0.45	0.44	0.45	0.43	0.41
AES	Encryption	0.66	0.6	0.63	0.63	0.62	0.63
	Decryption	0.66	0.71	0.64	0.64	0.19	0.19
Blowfish	Encryption	37.07	32	19	13	15	14
	Decryption	0.81	0.77	0.81	0.58	0.74	0.59
3DES	Encryption	14	11	17.7	10.21	10.11	13
	Decryption	0.77	0.79	0.78	0.6	0.6	0.6
BIO sym. algorithm	Encryption	1896	1848	1857	1846	1850	1850
	Decryption	2.62	13.1	1.83	1.31	1.57	2.62

In Figure 5 and 6, we will illustrate the maximum, mean, olimpic (by eliminating the absolute minimum and maximum values) and minimum encryption and decryption time for the Symmetric Bio Algorithm.

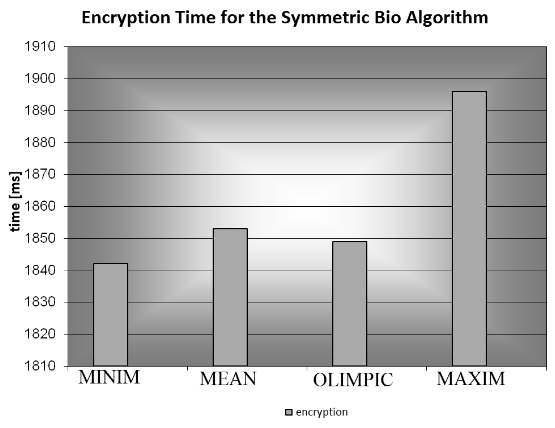


Figure 5. Encryption time for the Symmetric Bio Algorithm

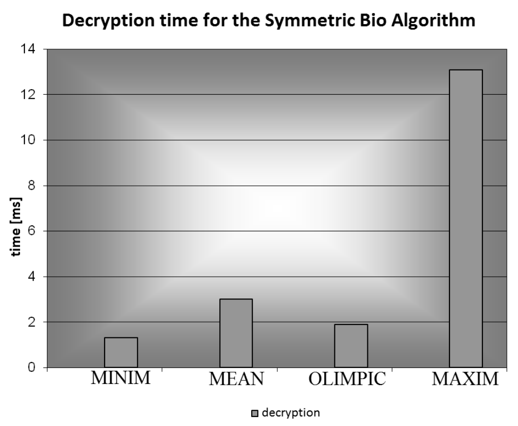


Figure 6. Decryption time for the Symmetric Bio Algorithm

First of all, we can notice that the systems 1 and 2 (with Windows OS) have larger time variations for the encryption and decryption processes. The third system, based on the Linux platform, offers a better stability, since the variation of the execution time is smaller.

As seen from the figures and tables above, the DNA Cipher requires a longer execution time for encryption and decryption, comparatively to the other ciphers. We would expect these results because of the type conversions which are needed in the case of the symmetric Bio algorithm. All classical encryption algorithms process array of bytes while the DNA Cipher is about strings. The additional conversions from string to array of bytes and back make this cipher to require more time for encryption and decryption then other classic algorithms.

However, this inconvenience should be solved with the implementation of full DNA algorithms and the usage of Bio-processors, which would make use of the parallel processing power of DNA algorithms.

In this paper we proposed an asymmetric DNA mechanism that is more reliable and more powerful than the OTP DNA symmetric algorithm. As future developments, we would like to make some test for the asymmetric DNA algorithm and increase its execution time.

Acknowledgments. This work was supported by CNCSIS–UEFISCSU, project number PNII – IDEI 1083/2007-2010.

6 REFERENCES

1. Hook, D.**,** *Beginning Cryptography with Java*, Wrox Press, (2005)
2. Kahn, D., The codebrakers McMillan, New York, (1967)
3. Schneier, B., Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish): Springer-Verlag, andf, Fast Software Encryption, Cambridge Security Workshop Proceedings (1993).
4. Schena, M., *Microarray analysis* Wiley-Liss, July (2003)
5. Gehani, A., LaBean, T., Reif, J., *DNA-Based Cryptography.* s.l.: DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 54, and *Lecture Notes in Computer Science, Springer*, (2004)
6. Techateerawat, P., A Review on Quantum Cryptography Technology, International Transaction Journal of Engineering, Management & Applied Sciences & Technologies, Vol. 1, pp. 35-41, (2010)
7. Adleman, L. M., Molecular computation of solution to combinatorial problems, *Science, 266*, 1021-1024, (1994)
8 8. Genetics Home Reference. U.S. National Library of Medicine. http://ghr.nlm.nih.gov/handbook/basics/dna. (2011)
9. DNA Alphabet. VSNS BioComputing Division, http://www.techfak.uni-bielefeld.de/bcd/Curric/PrwAli/node7.html#SECTION00071000000000000000, (2011)
10. Schneier, B., Applied cryptography: protocols, algorithms, and source code in C, John Wiley & Sons Inc, (1996)
11. Amin, S. T., Saeb, M., El-Gindi, S., A DNA-based Implementation of YAEA Encryption Algorithm, IASTED International Conference on Computational Intelligence, San Francisco, pp. 120-125, (2006)

12. Java Cryptography Architecture. Sun Microsystems. http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html (2011)
13. Tornea, O., Borda, M., Hodorogea, T., Vaida, M.F., Encryption System with Indexing DNA Chromosomes Cryptographic Algorithm, IASTED International Conference on Biomedical Engineering (BioMed 2010), 15-18 Feb., Innsbruck, Austria, paper 680-099, pp. 12-15, (2010)
14. Vaida, M.F., Terec, R., Tornea, O., Chiorean, L., Vanea, A., DNA Alternative Security, Advances in Intelligent Systems and Technologies Proceedings ECIT2010 – 6th European Conference on Intelligent Systems and Technologies, Iasi, Romania, October 07-09, pp. 1-4, (2010)
15. Vaida, M.F., Terec, R., Alboaie, L., Alternative DNA Security using BioJava, DICTAP2011, Conference SDIWC, Univ. de Bourgogne, Dijon, France, 21-23 June, 2011, pp.455-469
16. Wilson, R. K., The sequence of Homo sapiens FOSMID clone ABC14-50190700J6, submitted to http://www.ncbi.nlm.nih.gov, (2009)
17. Holland, R.C.G., Down, T., Pocock, M., Prlić, A., Huen, D., James, K., Foisy, S., Dräger, A., Yates, A., Heuer, M., Schreiber M.J., "BioJava: an Open-Source Framework for Bioinformatics", Bioinformatics (2008)
18. Hodorogea, T., Vaida, M. F., Deriving DNA Public Keys from Blood Analysis, International Journal of Computers Communications & Control Volume: 1 Pages: 262-267, (2006)
19. Wagner, N. R., The Laws of Cryptography with Java Code. [PDF], (2003).
20. BioJava – http://java.sun.com/developer/technicalArticles/javaopensource/biojava/, (2011)
21. RSA Security Inc. Public-Key Cryptography Standards (PKCS) – "PKCS #5 v2.0: Password-Based Cryptography Standard", (2000)
22. Nobelis, N., Boudaoud K., Riveill M., "Une architecture pour le transfert électronique sécurisé de document", PhD Thesis, Equipe Rainbow, Laboratories I3S – CNRS, Sophia-Antipolis, France, (2008)