

State Committee of communication, information and telecommunication
technologies
Tashkent University of Information Technologies

Faculty: CIF
Direction: Information Security

Coursework

Theme:

Protocols of the distribution of cryptography keys and
digital signature protocols

Completed by: M. Salimov
student group 233-12

Checked by: _____

- Tashkent in 2014 -

Content

Introduction.....	3
Chapter I. Protocols of the distribution of cryptographic keys.....	5
1.1 Cryptographic protocol.....	5
1.2 Cipher.....	9
1.3 Symmetric ciphers.....	10
1.4 Asymmetric ciphers.....	10
1.5 Data Encryption Standard (DES).....	11
1.6 Rivest, Shamir and Adleman (RSA).....	16
Chapter II. Digital signature protocols.....	20
2.1 Digital Signature.....	20
2.2 Hash function.....	22
Conclusion.....	25
References.....	26
Appendix.....	27

Introduction

Cryptography is a method of storing and transmitting data in a particular form so that only those for whom it is intended can read and process it. The term is most often associated with scrambling plaintext (ordinary text, sometimes referred to as cleartext) into ciphertext (a process called encryption), then back again (known as decryption).

Cryptography is closely related to the disciplines of cryptology and cryptanalysis. Cryptography includes techniques such as microdots, merging words with images, and other ways to hide information in storage or transit. However, in today's computer-centric world, cryptography is most often associated with scrambling plaintext (ordinary text, sometimes referred to as cleartext) into ciphertext (a process called encryption), then back again (known as decryption). Individuals who practice this field are known as cryptographers.

Modern cryptography concerns itself with the following four objectives:

- 1) **Confidentiality** (the information cannot be understood by anyone for whom it was unintended)
- 2) **Integrity** (the information cannot be altered in storage or transit between sender and intended receiver without the alteration being detected)
- 3) **Non-repudiation** (the creator/sender of the information cannot deny at a later stage his or her intentions in the creation or transmission of the information)
- 4) **Authentication** (the sender and receiver can confirm each other's identity and the origin/destination of the information)

Procedures and protocols that meet some or all of the above criteria are known as cryptosystems. Cryptosystems are often thought to refer only to mathematical procedures and computer programs; however, they also include the regulation of human behavior, such as choosing hard-to-guess passwords, logging off unused systems, and not discussing sensitive procedures with outsiders.

The word is derived from the Greek *kryptos*, meaning hidden. The origin of cryptography is usually dated from about 2000 BC, with the Egyptian practice of hieroglyphics. These consisted of complex pictograms, the full meaning of which

was only known to an elite few. The first known use of a modern cipher was by Julius Caesar (100 BC to 44 BC), who did not trust his messengers when communicating with his governors and officers. For this reason, he created a system in which each character in his messages was replaced by a character three positions ahead of it in the Roman alphabet.

In recent times, cryptography has turned into a battleground of some of the world's best mathematicians and computer scientists. The ability to securely store and transfer sensitive information has proved a critical factor in success in war and business.

Because governments do not wish certain entities in and out of their countries to have access to ways to receive and send hidden information that may be a threat to national interests, cryptography has been subject to various restrictions in many countries, ranging from limitations of the usage and export of software to the public dissemination of mathematical concepts that could be used to develop cryptosystems. However, the Internet has allowed the spread of powerful programs and, more importantly, the underlying techniques of cryptography, so that today many of the most advanced cryptosystems and ideas are now in the public domain.

Chapter I. Protocols of the distribution of cryptographic keys

Cryptographic protocol

Cryptographic protocol (Eng. Cryptographic protocol) - is an abstract or concrete protocol that includes a set of cryptographic algorithms. The basis of the protocol is a set of rules governing the use of cryptographic algorithms and transformations in information processes.

The functions of cryptographic protocols

- Data origin authentication
- Authentication of the parties
- Data Privacy
- Non-repudiation
- Non-repudiation with proof of receipt
- Non-repudiation with proof of source
- Data integrity
- Ensuring the integrity of the connection without recovery
- Ensuring the integrity of the connection with the restoration
- Access control

Classification

- Protocols of encryption / decryption
- Protocols of the electronic digital signature (EDS)
- Protocols of identification / authentication
- Authenticated key distribution protocols

Protocols encrypt / decrypt the basis of this protocol class contains some symmetrical or asymmetrical encryption algorithm / decryption. The encryption algorithm is executed to transfer the message sender, whereby an open connection is converted from a form encrypted. Decryption algorithm is performed on the reception by the recipient, whereby the encrypted message is converted from the open mold. This ensures the confidentiality of the property.

To ensure the integrity of the properties of the transmitted messages symmetric encryption / decryption, usually combined with an algorithm for calculating insert (WPI) for transmission and reception on check WPI, which uses an encryption key. When using asymmetric encryption algorithms / decryption wholeness provided separately by calculating the electronic digital signature (EDS) for transmission and digital signature verification at the reception, which ensures the properties of reliability and authenticity of the received message.

Protocols of the electronic digital signature (EDS) At the heart of the protocol of this class contains an algorithm for calculating the electronic signature on the transfer using the sender's private key and digital signature verification at the reception with the corresponding public key extracted from the open directory, but protected from modification. In case of a positive test result report is usually completed backup operations of the received message, its digital signature and the corresponding public key. Archive operation can not be performed if the electronic signature is used only to ensure the integrity and authenticity of the properties of the received message, but not non-repudiation. In this case, after verification, digital signature can be deleted immediately or after a limited period of time-out.

Protocols of identification / authentication

At the core authentication protocol contains an algorithm checks the fact that an identifiable object (user, device, process, ...), to present a name (Cipher), knows the secret information known only to the claimed subject matter, and verification method is, of course, indirectly, the there is no presentation of this sensitive information.

Usually with each name (Cipher) of the object is associated list of its rights and powers in the system, recorded in a secure database. In this case, the authentication protocol can be extended to the authentication protocol in which the identified object is checked for eligibility to order services.

If the authentication protocol used by EDS, the role of secret information plays a secret key electronic signature and electronic signature verification is performed using the public key digital signature, knowledge of which is impossible

to determine the corresponding private key, but makes sure that it is known to the author of EDS.

Authenticated key distribution protocols

Protocols of this class combine user authentication protocol generation and key distribution over a communication channel. The protocol has two or three members; third party is the center of the generation and distribution of keys (TSGRK), called for brevity server S. The protocol consists of three phases, with the names: generation, registration and communication. Generating the server S generates numerical values of system parameters, including a secret key and a public key. At the stage of registration server S identifies users on documents (for personal appearance or through authorized persons) for each object generates a key and / or identification information and generates a security token containing the necessary system constants and the public key of the server S (if necessary). At the stage of communication is implemented properly authenticated key exchange protocol, which completes the formation of a common session key.

Tasks

- Providing various authentication modes
- Generation, distribution and approval of cryptographic keys
- Protection interactions of participants
- The division of responsibilities between the parties

Variety of attacks on protocols

- Attacks against cryptographic algorithms
- Attacks against cryptographic methods used to implement protocols
- Attacks against the protocols themselves (active or passive)

Requirements for the safety protocol

1. Authentication (non-broadcast):

- authentication entity
- message Authentication
- replay protection

2. Authentication when sending to multiple locations or when connecting to the subscription / notification:

- implicit (hidden) authentication recipient
- origin authentication

3. Authorization (trusted third party)

4. Properties together key generation:

- authentication key
- validation key
- protection from reading ago
- the formation of new keys
- protected by the opportunity to negotiate security settings

5. Confidentiality

6. Anonymity:

- protection Ciphers listening (unlink ability)
- protection Ciphers from other members

7. Limited protection against attacks such as "denial of service"

8. Invariance of the sender

9. Non-repudiation of the earlier acts committed:

- accountability
- proof of the source
- proof of the recipient

10. Safe temporary property

Cipher

Cipher - any text conversion system with a secret (key) to ensure the privacy of information transmitted. Ciphers are used for secret correspondence with diplomatic representatives of their governments, in the armed forces for the text of secret documents on the technical means of communication, banks, security of transactions, as well as some Internet services for various reasons.

Cipher can be a set of symbols (conditional alphabet of numbers or letters) or an algorithm for transforming ordinary numbers and letters. The process of classifying messages using encryption cipher called. The science of creating and using ciphers called cryptography. Cryptanalysis - the science of methods for obtaining the initial value of encrypted information.

An important parameter of any cipher is key - parameter of cryptographic algorithm, which provides a choice of one of the plurality of conversion transformations possible for this algorithm. In modern cryptography is assumed that all the secrecy of the cryptographic algorithm is concentrated in the key, but not the details of the algorithm (Kerckhoffs's principle).

Not to be confused with the coding cipher - a fixed conversion of information from one form to another. In the latter there is no concept of the key and not performed Kerckhoffs's principle. In our time, the coding is almost never used to protect information from unauthorized access, but only on the transmission errors (noiseless coding) and other purposes not related to protection.

Types of ciphers

Ciphers may use the same key for encryption and decryption, or two different keys. On this basis distinguish:

- Uses a symmetric cipher key to encrypt and decrypt.
- Asymmetric cipher uses two different keys.

Ciphers can be designed to either encrypt the entire text at once, or encrypt it as they become available. Thus, there are:

- A block cipher encrypts the entire block of text at once, giving the ciphertext after receipt of all information.

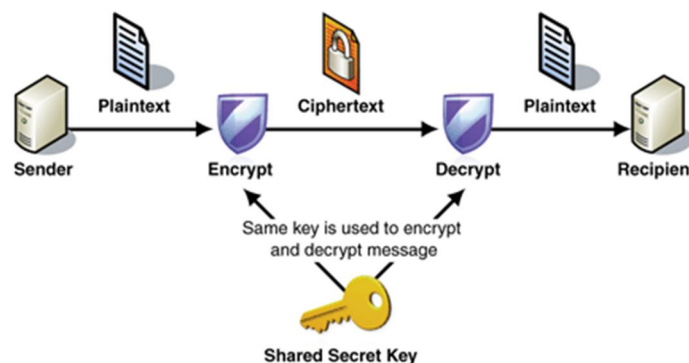
- A stream cipher encrypts the information and gives the ciphertext as they become available, thus having the ability to handle unlimited text size, using a fixed amount of memory.

Naturally, the block cipher that can be converted in-line, dividing the input data into blocks and encoding them separately.

There are also now unused substitution ciphers that have (for the most part) cryptographically weak.

Symmetric ciphers

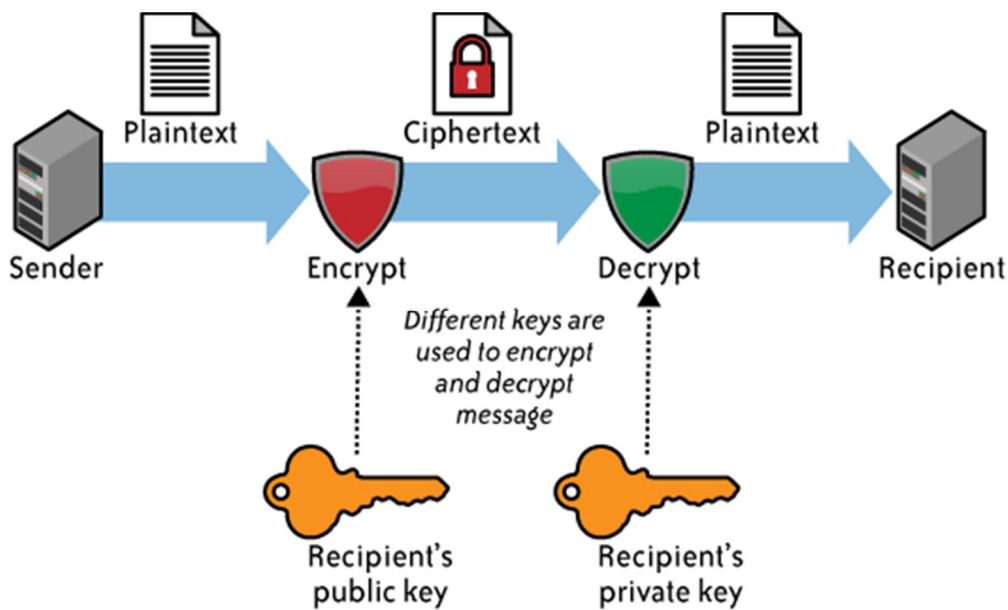
Symmetric cryptosystems (also symmetric encryption, symmetric ciphers) - encryption method in which to encrypt and decrypt apply the same cryptographic key. Before the invention of asymmetric encryption schemes only there were a way is a symmetric encryption. Key algorithm should be kept secret by both sides. Encryption algorithm selected by the parties prior to the exchange of messages.



Asymmetric ciphers

Asymmetric cipher - a system of encryption and / or digital signatures (EDS), where the public key is transmitted via an open (ie, insecure, available for follow-up) channel, and is used to verify the digital signature and to encrypt the message. To generate the digital signature and to decrypt the message using a secret key. [1] The cryptographic public key systems are now widely used in a variety of network protocols, in particular in the TLS and its predecessor SSL (underlying HTTPS), in SSH. Also used in PGP, S / MIME.

- RSA
- Elgamal
- Elliptic curve cryptography, ECC - (cryptosystem based on elliptic curves)



Data Encryption Standard

The **Data Encryption Standard (DES)** was once a predominant symmetric-key algorithm for the encryption of electronic data. It was highly influential in the advancement of modern cryptography in the academic world. Developed in the early 1970s at IBM and based on an earlier design by Horst Feistel, the algorithm was submitted to the National Bureau of Standards (NBS) following the agency's invitation to propose a candidate for the protection of sensitive, unclassified electronic government data. In 1976, after consultation with the National Security Agency (NSA), the NBS eventually selected a slightly modified version (strengthened against differential cryptanalysis, but weakened against brute force attacks), which was published as an official Federal Information Processing Standard (FIPS) for the United States in 1977. The publication of an NSA-approved encryption standard simultaneously resulted in its quick international adoption and widespread academic scrutiny. Controversies arose out of classified design elements, a relatively short key length of the symmetric-key block cipher design, and the involvement of the NSA, nourishing suspicions about a backdoor. The intense academic scrutiny the algorithm received over time led to the modern understanding of block ciphers and their cryptanalysis.

DES is now considered to be insecure for many applications. This is chiefly due to the 56-bit key size being too small; in January, 1999, distributed.net and the Electronic Frontier Foundation collaborated to publicly break a DES key in 22 hours and 15 minutes (see chronology). There are also some analytical results which demonstrate theoretical weaknesses in the cipher, although they are infeasible to mount in practice. The algorithm is believed to be practically secure in the form of Triple DES, although there are theoretical attacks. In recent years, the cipher has been superseded by the Advanced Encryption Standard (AES). Furthermore, DES has been withdrawn as a standard by the National Institute of Standards and Technology (formerly the National Bureau of Standards).

Some documentation makes a distinction between DES as a standard and DES as an algorithm, referring to the algorithm as the **DEA (Data Encryption Algorithm)**.

Description

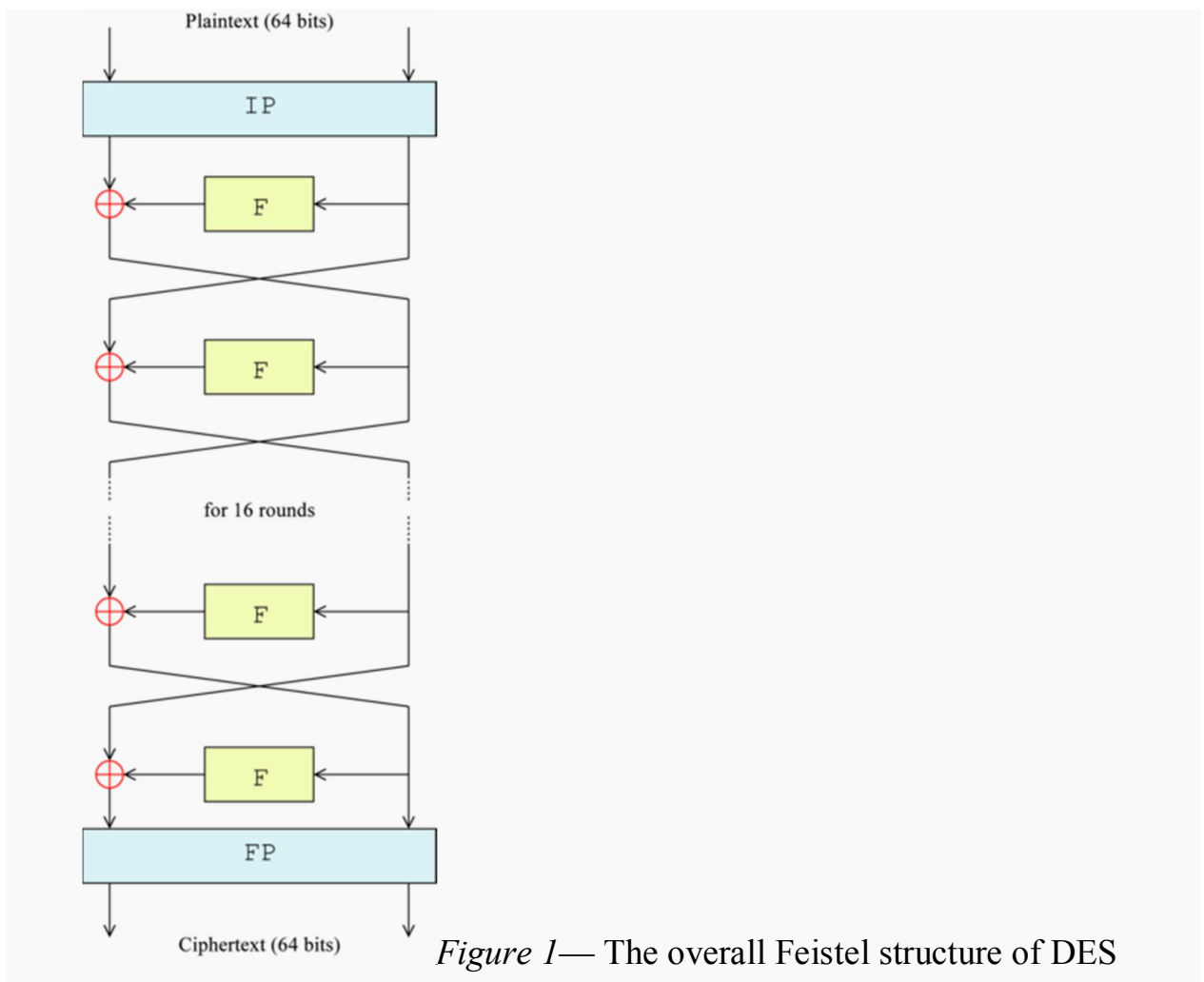


Figure 1— The overall Feistel structure of DES

For brevity, the following description omits the exact transformations and permutations which specify the algorithm; for reference, the details can be found in DES supplementary material.

DES is the archetypal block cipher — an algorithm that takes a fixed-length string of plaintext bits and transforms it through a series of complicated operations into another ciphertext bit string of the same length. In the case of DES, the block size is 64 bits. DES also uses a key to customize the transformation, so that decryption can supposedly only be performed by those who know the particular key used to encrypt. The key ostensibly consists of 64 bits; however, only 56 of these are actually used by the algorithm. Eight bits are used solely for checking parity, and are thereafter discarded. Hence the effective key length is 56 bits, and it is always quoted as such.


The key is nominally stored or transmitted as 8 bytes, each with odd parity. According to ANSI X3.92-1981 (Now, known as ANSIINCITS 92-1981), section 3.5:

One bit in each 8-bit byte of the *KEY* may be utilized for error detection in key generation, distribution, and storage. Bits 8, 16,..., 64 are for use in ensuring that each byte is of odd parity.

Like other block ciphers, DES by itself is not a secure means of encryption but must instead be used in a mode of operation. FIPS-81 specifies several modes for use with DES. Further comments on the usage of DES are contained in FIPS-74.

Decryption uses the same structure as encryption but with the keys used in reverse order. (This has the advantage that the same hardware or software can be used in both directions.)

Overall structure

 This section **does not cite any references or sources**. Please help improve this section by adding citations to reliable sources. Unsourced material may

be challenged and removed. (August 2009)

The algorithm's overall structure is shown in Figure 1: there are 16 identical stages of processing, termed *rounds*. There is also an initial and final permutation, termed *IP* and *FP*, which are inverses (*IP* "undoes" the action of *FP*, and vice versa). *IP* and *FP* have no cryptographic significance, but were included in order to facilitate loading blocks in and out of mid-1970s 8-bit based hardware.^[21]

Before the main rounds, the block is divided into two 32-bit halves and processed alternately; this criss-crossing is known as the Feistel scheme. The Feistel structure ensures that decryption and encryption are very similar processes — the only difference is that the sub keys are applied in the reverse order when decrypting. The rest of the algorithm is identical. This greatly simplifies implementation, particularly in hardware, as there is no need for separate encryption and decryption algorithms.

The \oplus symbol denotes the exclusive-OR (XOR) operation. The *F-function* scrambles half a block together with some of the key. The output from the *F-function* is then combined with the other half of the block, and the halves are swapped before the next round. After the final round, the halves are swapped; this is a feature of the Feistel structure which makes encryption and decryption similar processes.

The Feistel (F) function

The *F-function*, depicted in Figure 2, operates on half a block (32 bits) at a time and consists of four stages:

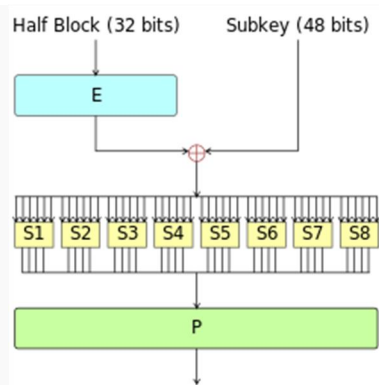


Figure 2—The Feistel function (F-function) of DES

1. *Expansion* — the 32-bit half-block is expanded to 48 bits using the *expansion permutation*, denoted E in the diagram, by duplicating half of the bits. The output consists of eight 6-bit ($8 * 6 = 48$ bits) pieces, each containing a copy of 4 corresponding input bits, plus a copy of the immediately adjacent bit from each of the input pieces to either side.

2. *Key mixing* — the result is combined with a *sub key* using an XOR operation. 16 48-bit sub keys — one for each round — are derived from the main key using the *key schedule* (described below).

3. *Substitution* — after mixing in the sub key, the block is divided into eight 6-bit pieces before processing by the *S-boxes*, or *substitution boxes*. Each of the eight S-boxes replaces its six input bits with four output bits according to a non-linear transformation, provided in the form of a lookup table. The S-boxes provide the core of the security of DES — without them, the cipher would be linear, and trivially breakable.

4. *Permutation* — finally, the 32 outputs from the S-boxes are rearranged according to a fixed permutation, the *P-box*. This is designed so that, after permutation, each S-box's output bits are spread across 4 different S boxes in the next round.

The alternation of substitution from the S-boxes, and permutation of bits from the P-box and E-expansion provides so-called "confusion and diffusion" respectively, a concept identified by Claude Shannon in the 1940s as a necessary condition for a secure yet practical cipher.

Key schedule

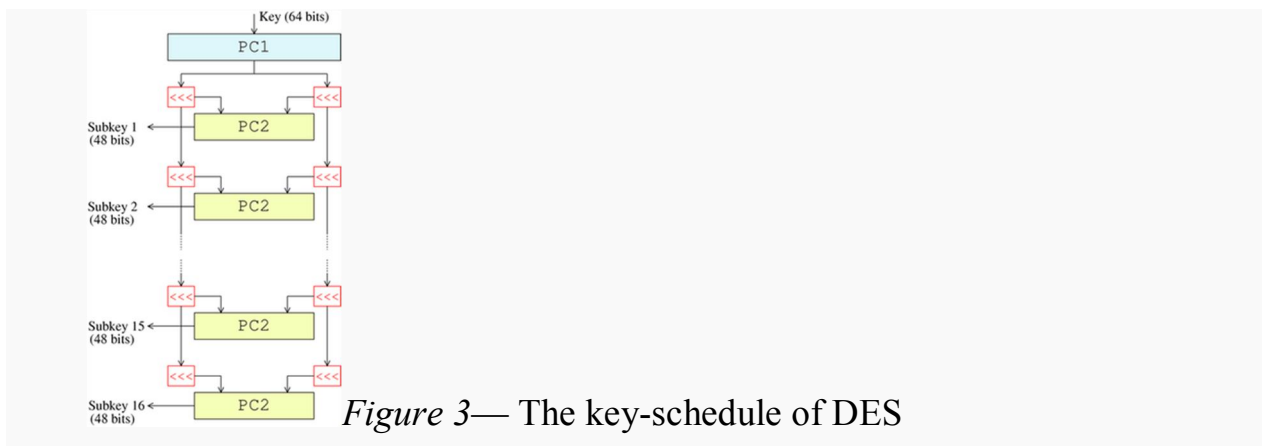


Figure 3— The key-schedule of DES

Figure 3 illustrates the *key schedule* for encryption — the algorithm which generates the sub keys. Initially, 56 bits of the key are selected from the initial 64 by *Permuted Choice 1 (PC-1)* — the remaining eight bits are either discarded or used as parity check bits. The 56 bits are then divided into two 28-bit halves; each half is thereafter treated separately. In successive rounds, both halves are rotated left by one or two bits (specified for each round), and then 48 sub key bits are selected by *Permuted Choice 2 (PC-2)* — 24 bits from the left half, and 24 from the right. The rotations (denoted by "<<<<" in the diagram) mean that a different set of bits is used in each sub key; each bit is used in approximately 14 out of the 16 sub keys.

The key schedule for decryption is similar — the sub keys are in reverse order compared to encryption. Apart from that change, the process is the same as for encryption. The same 28 bits are passed to all rotation boxes.

Security and cryptanalysis

Although more information has been published on the cryptanalysis of DES than any other block cipher, the most practical attack to date is still a brute force approach. Various minor cryptanalytic properties are known, and three theoretical attacks are possible which, while having a theoretical complexity less than a brute force attack, require an unrealistic number of known or chosen plaintexts to carry out, and are not a concern in practice.

RSA

RSA is one of the first practicable public-key cryptosystems and is widely used for secure data transmission. In such a cryptosystem, the encryption key is public and differs from the decryption key which is kept secret. In RSA, this asymmetry is based on the practical difficulty of factoring the product of two large prime numbers, the factoring problem. RSA stands for Ron Rivest, Adi Shamir and Leonard Adleman, who first publicly described the algorithm in 1977. Clifford Cocks, an English mathematician, had developed an equivalent system in 1973, but it was not declassified until 1997.

A user of RSA creates and then publishes a public key based on the two large prime numbers, along with an auxiliary value. The prime numbers must be kept secret. Anyone can use the public key to encrypt a message, but with currently published methods, if the public key is large enough, only someone with knowledge of the prime numbers can feasibly decode the message. Breaking RSA encryption is known as the RSA problem. It is an open question whether it is as hard as the factoring problem.

Operation

The RSA algorithm involves three steps: key generation, encryption and decryption.

Key generation

RSA involves a *public key* and a *private key*. The public key can be known by everyone and is used for encrypting messages. Messages encrypted with the public key can only be decrypted in a reasonable amount of time using the private key. The keys for the RSA algorithm are generated the following way:

1. Choose two distinct prime numbers p and q .
 - For security purposes, the integers p and q should be chosen at random, and should be of similar bit-length. Prime integers can be efficiently found using a primality test.
2. Compute $n = pq$.
 - n is used as the modulus for both the public and private keys. Its length, usually expressed in bits, is the key length.
3. Compute $\varphi(n) = \varphi(p)\varphi(q) = (p - 1)(q - 1) = n - (p + q - 1)$, where φ is Euler's totient function.
4. Choose an integer e such that $1 < e < \varphi(n)$ and $\text{gcd}(e, \varphi(n)) = 1$; i.e., e and $\varphi(n)$ are coprime.
 - e is released as the public key exponent.
 - e having a short bit-length and small Hamming weight results in more efficient encryption – most commonly $2^{16} + 1 = 65,537$. However, much smaller values of e (such as 3) have been shown to be less secure in some settings.^[5]

5. Determine d as $d \equiv e^{-1} \pmod{\varphi(n)}$; i.e., d is the multiplicative inverse of e (modulo $\varphi(n)$).

- This is more clearly stated as: solve for d given $d \cdot e \equiv 1 \pmod{\varphi(n)}$

- This is often computed using the extended Euclidean algorithm. Using the pseudo code in the *Modular integers* section, inputs a and n correspond to e and $\varphi(n)$, respectively.

- d is kept as the private key exponent.

The *public key* consists of the modulus n and the public (or encryption) exponent e . The *private key* consists of the modulus n and the private (or decryption) exponent d , which must be kept secret. p , q , and $\varphi(n)$ must also be kept secret because they can be used to calculate d .

- An alternative, used by PKCS#1, is to choose d matching $de \equiv 1 \pmod{\lambda}$ with $\lambda = \text{lcm}(p-1, q-1)$, where lcm is the least common multiple. Using λ instead of $\varphi(n)$ allows more choices for d . λ can also be defined using the Carmichael function, $\lambda(n)$.

- The ANSI X9.31 standard prescribes, IEEE 1363 describes, and PKCS#1 allows, that p and q match additional requirements: being strong primes, and being different enough that Fermat factorization fails.

Encryption

Alice transmits her public key (n, e) to Bob and keeps the private key d secret. Bob then wishes to send message M to Alice.

He first turns M into an integer m , such that $0 \leq m < n$ by using an agreed-upon reversible protocol known as a padding scheme. He then computes the ciphertext c corresponding to

$$c \equiv m^e \pmod{n}$$

This can be done efficiently, even for 500-bit numbers, using Modular exponentiation. Bob then transmits c to Alice.

Note that at least nine values of m will yield a ciphertext c equal to m ,^[note 1] but this is very unlikely to occur in practice.

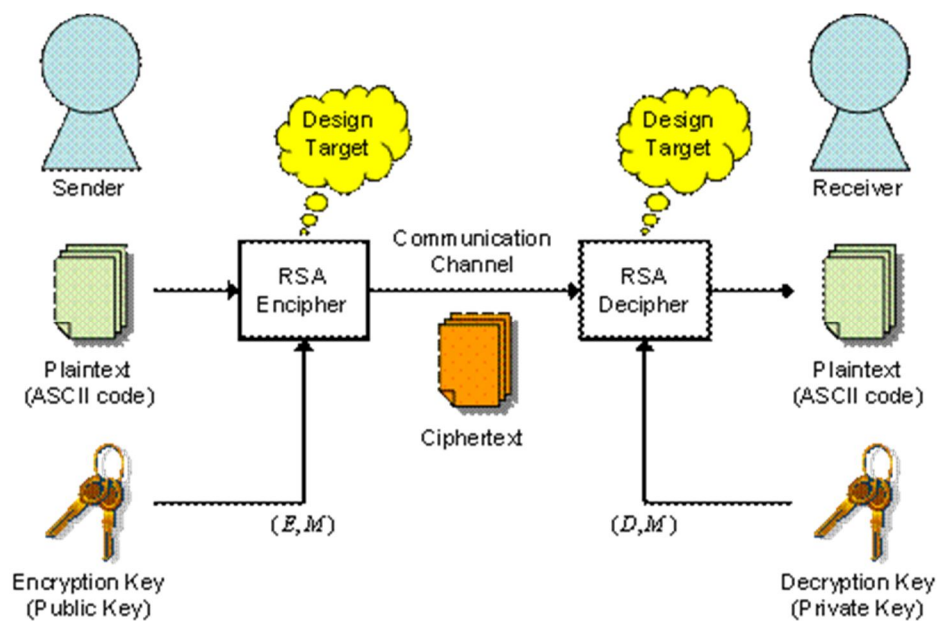
Decryption

Alice can recover m from c by using her private key exponent d via computing

$$m \equiv c^d \pmod{n}$$

Given m , she can recover the original message M by reversing the padding scheme.

(In practice, there are more efficient methods of calculating c^d using the precomputed values below.)



Chapter II. Digital signature protocols

Digital Signature

A **digital signature** is a mathematical scheme for demonstrating the authenticity of a digital message or document. A valid digital signature gives a recipient reason to believe that the message was created by a known sender, such that the sender cannot deny having sent the message (authentication and non-repudiation) and that the message was not altered in transit (integrity). Digital signatures are commonly used for software distribution, financial transactions, and in other cases where it is important to detect forgery or tampering.

Explanation

Digital signatures are often used to implement electronic signatures, a broader term that refers to any electronic data that carries the intent of a signature,^[1] but not all electronic signatures use digital signatures. In some countries, including the United States, India, Brazil,^[4] and members of the European Union, electronic signatures have legal significance.

Digital signatures employ a type of asymmetric cryptography. For messages sent through a no secure channel, a properly implemented digital signature gives the receiver reason to believe the message was sent by the claimed sender. In many instances, common with Engineering companies for example, digital seals are also required for another layer of validation and security. Digital seals and signatures are equivalent to handwritten signatures and stamped seals.^[5] Digital signatures are equivalent to traditional handwritten signatures in many respects, but properly implemented digital signatures are more difficult to forge than the handwritten type. Digital signature schemes, in the sense used here, are cryptographically based, and must be implemented properly to be effective. Digital signatures can also provide non-repudiation, meaning that the signer cannot successfully claim they did not sign a message, while also claiming their private key remains secret; further, some non-repudiation schemes offer a time stamp for the digital signature, so that even if the private key is exposed, the signature is valid. Digitally signed messages may be anything representable as a bit string: examples

include electronic mail, contracts, or a message sent via some other cryptographic protocol.

Definition

Main article: Public-key cryptography

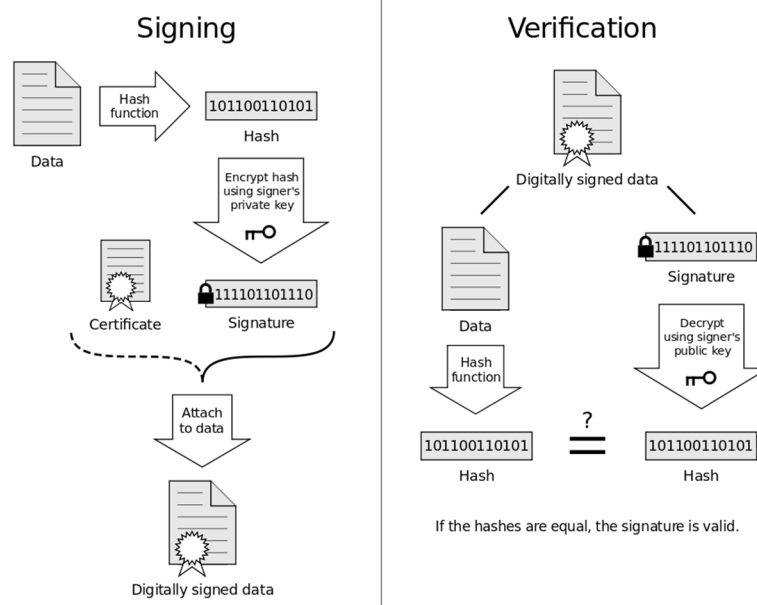
A digital signature scheme typically consists of three algorithms;

- A *key generation* algorithm that selects a *private key* uniformly at random from a set of possible private keys. The algorithm outputs the private key and a corresponding *public key*.

- A *signing* algorithm that, given a message and a private key, produces a signature.

- A *signature verifying* algorithm that, given a message, public key and a signature, either accepts or rejects the message's claim to authenticity.

Two main properties are required. First, the authenticity of a signature generated from a fixed message and fixed private key can be verified by using the corresponding public key. Secondly, it should be computationally infeasible to generate a valid signature for a party without knowing that party's private key. A digital signature is an authentication mechanism that enables the creator of message to attach a code that act as a signature. It is formed by taking the hash of message and encrypting the message with creator's private key.



A diagram showing how a digital signature is applied and then verified.

Hash function

A **cryptographic hash function** is a hash function which is considered practically impossible to invert, that is, to recreate the input data from its hash value alone. These one-way hash functions have been called "the workhorses of modern cryptography".^[1] The input data is often called the *message*, and the hash value is often called the *message digest* or simply the *digest*.

The ideal cryptographic hash function has four main properties:

- it is easy to compute the hash value for any given message
- it is infeasible to generate a message that has a given hash
- it is infeasible to modify a message without changing the hash
- it is infeasible to find two different messages with the same hash.

Cryptographic hash functions have many information security applications, notably in digital signatures, message authentication codes (MACs), and other forms of authentication. They can also be used as ordinary hash functions, to index data in hash tables, for fingerprinting, to detect duplicate data or uniquely identify files, and as checksums to detect accidental data corruption. Indeed, in information security contexts, cryptographic hash values are sometimes called (*digital*) *fingerprints*, *checksums*, or just *hash values*, even though all these terms stand for more general functions with rather different properties and purposes.

Hash functions based on block ciphers

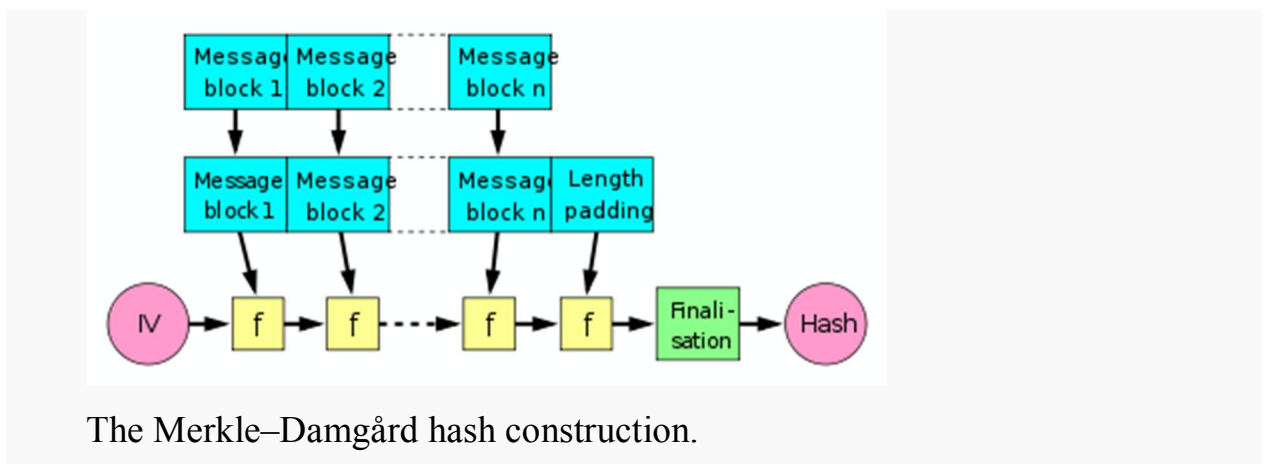
There are several methods to use a block cipher to build a cryptographic hash function, specifically a one-way compression function.

The methods resemble the block cipher modes of operation usually used for encryption. All well-known hash functions, including MD4, MD5, SHA-1 and SHA-2 are built from block-cipher-like components designed for the purpose, with feedback to ensure that the resulting function is not invertible. SHA-3 finalists included functions with block-cipher-like components (e.g., Skein, BLAKE) though the function finally selected, Keccak, was built on a cryptographic sponge instead.

A standard block cipher such as AES can be used in place of these custom block ciphers; that might be useful when an embedded system needs to implement both encryption and hashing with minimal code size or hardware area. However, that approach can have costs in efficiency and security. The ciphers in hash functions are built for hashing: they use large keys and blocks, can efficiently change keys every block, and have been designed and vetted for resistance to related-key attacks. General-purpose ciphers tend to have different design goals. In particular, AES has key and block sizes that make it nontrivial to use to generate long hash values; AES encryption becomes less efficient when the key changes each block; and related-key attacks make it potentially less secure for use in a hash function than for encryption.

Merkle-Damgård construction

Main article: Merkle–Damgård construction



The Merkle–Damgård hash construction.

A hash function must be able to process an arbitrary-length message into a fixed-length output. This can be achieved by breaking the input up into a series of equal-sized blocks, and operating on them in sequence using a one-way compression function. The compression function can either be specially designed for hashing or be built from a block cipher. A hash function built with the Merkle–Damgård construction is as resistant to collisions as is its compression function; any collision for the full hash function can be traced back to a collision in the compression function.

The last block processed should also be unambiguously length padded; this is crucial to the security of this construction. This construction is called the Merkle–

Damgård construction. Most widely used hash functions, including SHA-1 and MD5, take this form.

The construction has certain inherent flaws, including length-extension and generate-and-paste attacks, and cannot be parallelized. As a result, many entrants in the current NIST hash function competition are built on different, sometimes novel, constructions.

Conclusion

The above communication protocols to establish a common key provide great opportunities for the development of secure communication protocols for dynamic groups. On the basis of these you can build complex authentication protocols, digital signature, proof of knowledge.

There are many problems associated with signatures for the groups. First and foremost is the problem of resistance to attack users' associations and the problem of removing the band members and their keys. In the area of the proposed anonymity for participants in the group is a major problem. Also not completely defined intergroup interaction and cases with simultaneous membership in multiple groups (using multiple keys recognized irrational) and education subgroups. Existing schemes are preliminary.

Interaction protocols with network technologies. In view of the purely practical aspects of these materials are not taken into account.

The challenge now secure communication within a group dynamic composition of the participants received increased attention due to the widespread use of technologies of the Internet. Maybe soon there will be new key distribution protocols that do not contain deficiencies described protocols.

References

1. G. Ateniese, M. Steiner, G. Tsudik “Authenticated Group Key Agreement and Friends”, in *ACM Symposium on Computer and Communication Security*, November 1998.
2. M. Steiner, G. Tsudik, M. Waidner “Diffie-Hellman key distribution extended to groups”, in *ACM Conference on Computer and Communications Security*, pp.31-37, ACM Press, Mar. 1996.
3. G. Ateniese, D. Hasse, O. Chevassut, Y. Kim, G. Tsudik “The Design of a Group Key Agreement API”, IBM Research Division, Zurich Research Laboratory.
4. Y. Amir, G. Ateniese, D. Hasse, Y. Kim, C. Nita-Rotaru, T. Sclossnagle, J. Schultz, J. Stanton, G. Tsudik “Secure Group Communications in Asynchronous Networks with Failures: Integration and Experiments”, 1999.
5. G. Caronni, M. Waldvoget, D. Sun, B. Plattner “Efficient Security for Large and Dynamic Multicast Groups”, Computer Engineering and Networks Laboratory.
6. RSA Laboratories, <http://www.rsalab.com>
7. <http://en.wikipedia.org/>

Appendix

The program code in JAVA

DES class

```
package cryptprotocols;
/*
 * @author Mumin
 */
import javax.crypto.*;
import javax.crypto.spec.*;
class DesCipher {
    // Algorithm used
    private final static String ALGORITHM = "DES";
    /**
     * Encrypt data
     * @param secretKey - a secret key used for encryption
     * @param data - data to encrypt
     * @return Encrypted data
     * @throws Exception
     */
    public static String cipher(String secretKey, String data) throws Exception
    {
        // Key has to be of length 8
        if (secretKey == null || secretKey.length() != 8)
            throw new Exception("Invalid key length - 8 bytes key needed!");
        SecretKey key = new SecretKeySpec(secretKey.getBytes(),
ALGORITHM);
        Cipher cipher = Cipher.getInstance(ALGORITHM);
        cipher.init(Cipher.ENCRYPT_MODE, key);

        return toHex(cipher.doFinal(data.getBytes()));
    }
    /**
     * Decrypt data
     * @param secretKey - a secret key used for decryption
     * @param data - data to decrypt
     * @return Decrypted data
     * @throws Exception
     */
}
```

```

    public static String decipher(String secretKey, String data) throws
Exception {
    // Key has to be of length 8
    if (secretKey == null || secretKey.length() != 8)
        throw new Exception("Invalid key length - 8 bytes key needed!");

    SecretKey key = new SecretKeySpec(secretKey.getBytes(),
ALGORITHM);
    Cipher cipher = Cipher.getInstance(ALGORITHM);
    cipher.init(Cipher.DECRYPT_MODE, key);

    return new String(cipher.doFinal(toByte(data)));
}
// Helper methods
private static byte[] toByte(String hexString) {
    int len = hexString.length()/2;
    byte[] result = new byte[len];
    for (int i = 0; i < len; i++)
        result[i] = Integer.valueOf(hexString.substring(2*i, 2*i+2),
16).byteValue());
    return result;
}

public static String toHex(byte[] stringBytes) {
    StringBuffer result = new StringBuffer(2*stringBytes.length);

    for (int i = 0; i < stringBytes.length; i++) {
result.append(HEX.charAt((stringBytes[i]>>4)&0x0f)).append(HEX.charAt(string
Bytes[i]&0x0f));
    }
    return result.toString();
}
private final static String HEX = "0123456789ABCDEF";
// Helper methods - end
/*
 * Quick test
 * @param args
 */
}

```

RSA class

```
package cryptprotocols;
import com.sun.org.apache.xerces.internal.impl.dv.util.Base64;
import java.io.*;
import java.math.BigInteger;
import java.security.*;
import java.security.spec.*;
import java.util.Scanner;
import javax.crypto.*;
/*
 * @author Mumin
 */
class RSAencrypt{
    Cipher c;
    RSAencrypt() throws NoSuchAlgorithmException,
NoSuchPaddingException {
        c = Cipher.getInstance("RSA");
    }
    public String RSAencrypt(String plaintext, String file) throws
InvalidKeyException, IllegalBlockSizeException, BadPaddingException,
FileNotFoundException, NoSuchAlgorithmException, InvalidKeySpecException,
UnsupportedEncodingException {
        int k = 0;
        Scanner reader = new Scanner(new File(file));
        String[] str = new String[2];
        while (reader.hasNext()){
            str[k] = reader.next();k++;
        }

        BigInteger pub_m = new BigInteger(str[0]);
        BigInteger pub_x = new BigInteger(str[1]);

        KeyFactory keyFactory = KeyFactory.getInstance("RSA");
        RSAPublicKeySpec new_pubks = new RSAPublicKeySpec(pub_m, pub_x);
        PublicKey new_public = keyFactory.generatePublic(new_pubks);

        c.init(Cipher.ENCRYPT_MODE, new_public);
        byte[] encrypted = c.doFinal(plaintext.getBytes("utf8"));
```

```
String result = Base64.encode(encrypted);
return result;
}
```

```
public String RSAdecrypt(String ciphertext, String file) throws
FileNotFoundException, NoSuchAlgorithmException, InvalidKeySpecException,
InvalidKeyException, IllegalBlockSizeException, BadPaddingException,
UnsupportedEncodingException{
```

```
int k = 0;
Scanner reader = new Scanner(new File(file));
String[] str = new String[2];
while (reader.hasNext()){
    str[k] = reader.next();k++;
}
}
```

```
BigInteger pr_m = new BigInteger(str[0]);
BigInteger pr_x = new BigInteger(str[1]);
```

```
KeyFactory keyFactory = KeyFactory.getInstance("RSA");
RSAPrivateKeySpec new_prks = new RSAPrivateKeySpec(pr_m, pr_x);
PrivateKey new_private = keyFactory.generatePrivate(new_prks);
```

```
byte[] data1=Base64.decode(ciphertext);
c.init(Cipher.DECRYPT_MODE, new_private);
byte[] decrypted = c.doFinal(data1);
```

```
String result = new String(decrypted);
return result;
```

```
}
}
```

HASH function class

```
package cryptprotocols;
```

```
/*
```

```
* @author Mumin
```

```
*/
```

```
import java.io.UnsupportedEncodingException;
```

```
import java.math.BigInteger;
```

```
import java.security.*;
```

```

public class MD5Hash {

    public String getHash(String str) throws NoSuchAlgorithmException,
        UnsupportedEncodingException {

        MessageDigest m = MessageDigest.getInstance("MD5");
        m.reset();
        // передаем в MessageDigest байт-код строки
        m.update(str.getBytes("utf-8"));
        // получаем MD5-хеш строки без лидирующих нулей
        String s2 = new BigInteger(1, m.digest()).toString(16);
        StringBuilder sb = new StringBuilder(32);
        // дополняем нулями до 32 символов, в случае необходимости
        //System.out.println(32 - s2.length());
        for (int i = 0, count = 32 - s2.length(); i < count; i++) {
            sb.append("0");
        }
        // возвращаем MD5-хеш
        return sb.append(s2).toString();
    }
}

```

Main frame

```

package cryptprotocols;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
 * @author Mumin
 */
public class CryptProtocols extends JFrame implements ActionListener{
    JLabel label;
    JButton b1,b2;
    JRadioButton ch1,ch2,ch3;
    ButtonGroup radioButtonGroup;

    public static void main(String[] args) {
        // TODO code APPLICATION logic here
        CryptProtocols cr8 = new CryptProtocols();
    }
}

```

```

public CryptProtocols(){
    setLayout(new BorderLayout());
    setBackground(Color.WHITE);
    setSize(200, 175);
    Font appletFont = new Font("Monospaced", Font.BOLD, 20);
    this.setFont(appletFont);

    Panel panel=new Panel();
    panel.setLayout(new GridLayout(0,1));
    label = new JLabel("Choose the programm...");
    radioButtonGroup = new ButtonGroup();
    ch1 = new JRadioButton("DES");
    ch2 = new JRadioButton("RSA");
    ch3 = new JRadioButton("DigitalSignature");
    panel.add(label);
    panel.add(ch1);
    panel.add(ch2);
    panel.add(ch3);
    radioButtonGroup.add(ch1);
    radioButtonGroup.add(ch2);
    radioButtonGroup.add(ch3);
    this.add(panel, "Center");

    Panel centerPanel = new Panel();
    centerPanel.setLayout(new GridLayout(1,2));
    b1 = new JButton("Select");
    b1.addActionListener(this);
    b2 = new JButton("Exit");
    b2.addActionListener(this);
    centerPanel.add(b1, "West");
    centerPanel.add(b2, "East");
    this.add(centerPanel, "South");

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLocationRelativeTo(null);
    setVisible(true);
}
@Override
public void actionPerformed(ActionEvent e) {

```



```

        JButton button = (JButton) e.getSource();
        if (button==b1){
            if (ch1.isSelected()) {DES des = new DES();}
            else if (ch2.isSelected()) {RSA rsa = new RSA();}
            else if (ch3.isSelected()) {DS ds=new DS();}
            }
        else if (button==b2) {System.exit(0);
        }
    }
}

```

DES frame

```

package cryptprotocols;
import java.awt.*;
import java.awt.event.*;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.*;
/*
 * @author Mumin
 */

public class DES extends JFrame implements ActionListener{
    JLabel l1,l2,l3;
    JTextArea t1,t2;
    JTextField tf;
    JButton b1,b2;
    public DES(){
        setLayout(new BorderLayout());
        setBackground(Color.WHITE);
        setSize(600, 300);

        Panel panel = new Panel();
        panel.setLayout(new GridLayout());
        l1 = new JLabel("Plaintext...");
        l2 = new JLabel("Ciphertext...");
        panel.add(l1);
        panel.add(l2);
        this.add(panel, "North");
    }
}

```

```

Panel centerPanel = new Panel();
centerPanel.setLayout(new GridLayout());
t1 = new JTextArea();
t2 = new JTextArea();
t2.setBackground(Color.CYAN);
centerPanel.add(t1);
centerPanel.add(t2);
this.add(centerPanel, "Center");

```

```

Panel southPanel = new Panel();
southPanel.setLayout(new GridLayout(2,2));
l3 = new JLabel("Enter the 8-digit key: ");
tf = new JTextField();
b1 = new JButton("Encrypt");
b1.addActionListener(this);
b2 = new JButton("Dencrypt");
b2.addActionListener(this);
southPanel.add(l3);
southPanel.add(tf);
southPanel.add(b1);
southPanel.add(b2);
this.add(southPanel, "South");

```

```

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setLocationRelativeTo(null);
setVisible(true);
}

```

@Override

```

public void actionPerformed(ActionEvent e) {
    JButton button = (JButton) e.getSource();
    DesCipher ds = new DesCipher();

    if (button==b1) {
        try {
            t2.setText(ds.cipher(tf.getText(),t1.getText()));
        } catch (Exception ex) {
            Logger.getLogger(DES.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}

```

```

}
    else if (button==b2) {
        try {
            t2.setText(ds.decipher(tf.getText(), t1.getText()));
        } catch (Exception ex) {
            Logger.getLogger(DES.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
}
}
}

```

RSA frame

```

package cryptprotocols;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.math.BigInteger;
import java.security.*;
import java.security.spec.*;
import java.util.logging.*;
import javax.crypto.*;
import javax.swing.*;
/*
 * @author Mumin
 */
public class RSA extends JFrame implements ActionListener{
    public String dir_patch = "";
    JLabel l1,l2,l3;
    JTextArea t1,t2;
    JTextField tf1,tf2;
    JButton b1,b2,b3,b4,b5;
    public RSA(){
        setLayout(new BorderLayout());
        setBackground(Color.WHITE);
        setSize(600, 300);

        Panel panel = new Panel();
        panel.setLayout(new GridLayout());
        l1 = new JLabel("Plaintext...");
        l2 = new JLabel("Ciphertext...");

```

```
panel.add(l1);  
panel.add(l2);  
this.add(panel, "North");
```

```
Panel centerPanel = new Panel();  
centerPanel.setLayout(new GridLayout());  
t1 = new JTextArea();  
t2 = new JTextArea();  
t2.setBackground(Color.CYAN);  
centerPanel.add(t1);  
centerPanel.add(t2);  
this.add(centerPanel, "Center");
```

```
Panel southPanel = new Panel();  
southPanel.setLayout(new GridLayout(4,2));  
b3 = new JButton("Private key ");  
b3.addActionListener(this);  
b4 = new JButton("Public key ");  
b4.addActionListener(this);  
l3 = new JLabel("Press to generate keys: ");  
b5 = new JButton("Generate");  
b5.addActionListener(this);  
tf1 = new JTextField("Private key file");  
tf2 = new JTextField("Public key file");  
b1 = new JButton("Encrypt");  
b1.addActionListener(this);  
b2 = new JButton("Decrypt");  
b2.addActionListener(this);  
southPanel.add(b3);  
southPanel.add(b4);  
southPanel.add(l3);  
southPanel.add(b5);  
southPanel.add(tf1);  
southPanel.add(tf2);  
southPanel.add(b1);  
southPanel.add(b2);  
this.add(southPanel, "South");
```

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```

        setLocationRelativeTo(null);
        setVisible(true);
    }

```

@Override

```

public void actionPerformed(ActionEvent e) {
    JButton button = (JButton) e.getSource();
    if (button==b1) {try {
        RSAencrypt rsac = new RSAencrypt();
        try {
            try {
                t2.setText(rsac.RSAencrypt(t1.getText(), tf1.getText()));
            } catch (UnsupportedEncodingException ex) {
                Logger.getLogger(RSA.class.getName()).log(Level.SEVERE, null, ex);
            }
            } catch (InvalidKeyException ex) {
                Logger.getLogger(RSA.class.getName()).log(Level.SEVERE, null, ex);
            } catch (IllegalBlockSizeException ex) {
                Logger.getLogger(RSA.class.getName()).log(Level.SEVERE, null, ex);
            } catch (BadPaddingException ex) {
                Logger.getLogger(RSA.class.getName()).log(Level.SEVERE, null, ex);
            } catch (FileNotFoundException ex) {
                Logger.getLogger(RSA.class.getName()).log(Level.SEVERE, null, ex);
            } catch (InvalidKeySpecException ex) {
                Logger.getLogger(RSA.class.getName()).log(Level.SEVERE, null, ex);
            }
            } catch (NoSuchAlgorithmException ex) {
                Logger.getLogger(RSA.class.getName()).log(Level.SEVERE, null, ex);
            } catch (NoSuchPaddingException ex) {
                Logger.getLogger(RSA.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
    } else if (button==b2) {try {
        RSAencrypt rsac = new RSAencrypt();
        try {
            try {
                t2.setText(rsac.RSAdecrypt(t1.getText(), tf2.getText()));
            } catch (UnsupportedEncodingException ex) {
                Logger.getLogger(RSA.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
    }
}

```

```

        }
        } catch (InvalidKeyException ex) {
Logger.getLogger(RSA.class.getName()).log(Level.SEVERE, null, ex);
        } catch (IllegalBlockSizeException ex) {
Logger.getLogger(RSA.class.getName()).log(Level.SEVERE, null, ex);
        } catch (BadPaddingException ex) {
Logger.getLogger(RSA.class.getName()).log(Level.SEVERE, null, ex);
        } catch (FileNotFoundException ex) {
Logger.getLogger(RSA.class.getName()).log(Level.SEVERE, null, ex);
        } catch (InvalidKeySpecException ex) {
Logger.getLogger(RSA.class.getName()).log(Level.SEVERE, null, ex);
        }
    } catch (NoSuchAlgorithmException ex) {
Logger.getLogger(RSA.class.getName()).log(Level.SEVERE, null, ex);
    } catch (NoSuchPaddingException ex) {
Logger.getLogger(RSA.class.getName()).log(Level.SEVERE, null, ex);
    }
}
else if (button==b3) {getfile();tf1.setText(dir_patch);}
else if (button==b4) {getfile();tf2.setText(dir_patch);}
else if (button==b5) {try {
    try {
        generateKeys(tf1.getText(), tf2.getText());
    } catch (FileNotFoundException ex) {
Logger.getLogger(RSA.class.getName()).log(Level.SEVERE, null, ex);
    } catch (IOException ex) {
Logger.getLogger(RSA.class.getName()).log(Level.SEVERE, null, ex);
    }
    } catch (NoSuchAlgorithmException ex) {
Logger.getLogger(RSA.class.getName()).log(Level.SEVERE, null, ex);
    } catch (InvalidKeySpecException ex) {
Logger.getLogger(RSA.class.getName()).log(Level.SEVERE, null, ex);
    }
}
}
}

```

```

public void getfile(){
    JFileChooser fileChooser = new JFileChooser();

```

```

fileChooser.setSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
int ret = fileChooser.showDialog(this, "Open");
String patch = null;
if (ret == JFileChooser.APPROVE_OPTION)
{
    patch = fileChooser.getSelectedFile().getAbsolutePath();
    dir_patch = patch;
    final File folder = new File(patch);
}
}

public void generateKeys(String publicFile, String privateFile) throws
NoSuchAlgorithmException, InvalidKeySpecException, FileNotFoundException,
IOException{
    KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("RSA");
    KeyPair keyPair = keyPairGen.generateKeyPair();
    PrivateKey privateKey = keyPair.getPrivate();
    PublicKey publicKey = keyPair.getPublic();

    KeyFactory keyFactory = KeyFactory.getInstance("RSA");
    RSAPrivateKeySpec rSAPrivateKey = keyFactory.getKeySpec(privateKey,
RSAPrivateKeySpec.class);

    BigInteger pr_m = rSAPrivateKey.getModulus();
    BigInteger pr_x = rSAPrivateKey.getPrivateExponent();

    RSAPublicKeySpec rsaPublicKey = keyFactory.getKeySpec(publicKey,
RSAPublicKeySpec.class);

    BigInteger pub_m = rsaPublicKey.getModulus();
    BigInteger pub_x = rsaPublicKey.getPublicExponent();

    String pubf = "keys/"+tf1.getText()+".txt";
    String prf = "keys/"+tf2.getText()+".txt";

    File publicKeyFile = new File(pubf);
    File privateKeyFile = new File(prf);

    FileOutputStream fl = null;

```

```

FileOutputStream f2 = null;

f1 = new FileOutputStream(publicKeyFile);
f2 = new FileOutputStream(privateKeyFile);

f1.write(pub_m.toString().getBytes());
f1.write(" ".getBytes());
f1.write(pub_x.toString().getBytes());

f2.write(pr_m.toString().getBytes());
f2.write(" ".getBytes());
f2.write(pr_x.toString().getBytes());

tf1.setText(pubf);
tf2.setText(prf);
}
}
Digital Signature frame
package cryptprotocols;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
 * @author Mumin
 */
public class DS extends JFrame implements ActionListener{
    JButton b1,b2;

    public DS(){
        setLayout(new BorderLayout());
        setBackground(Color.WHITE);
        setSize(350, 70);

        Panel centerPanel = new Panel();
        centerPanel.setLayout(new GridLayout(1,2));
        b1 = new JButton("Get Digital Signature");
        b1.addActionListener(this);
        b2 = new JButton("Check Digital Signature");
        b2.addActionListener(this);

```



```

        centerPanel.add(b1);
        centerPanel.add(b2);
        this.add(centerPanel, "Center");

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        setVisible(true);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        JButton button = (JButton) e.getSource();
        if (button==b1) {DigitalSignature dig = new DigitalSignature();}
        else if (button==b2) {CheckDS ds = new CheckDS();}
    }
}

```

Check Digital Signature frame

```

package cryptprotocols;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.security.*;
import java.security.spec.InvalidKeySpecException;
import java.util.Scanner;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.crypto.BadPaddingException;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.NoSuchPaddingException;
import javax.swing.*;
/*
 * @author Mumin
 */
public class CheckDS extends JFrame implements ActionListener{
    public String dir_patch = "";
    JLabel l1,l2,l3;
    JButton b1,b2,b3,b4;
    JTextField tf1;

```

```

public CheckDS(){
    setLayout(new BorderLayout());
    setBackground(Color.WHITE);
    setSize(400, 150);

    Panel centerPanel = new Panel();
    centerPanel.setLayout(new GridLayout(4,2));
    l1 = new JLabel("Document for cheking...");
    l2 = new JLabel("Public key...");
    l3 = new JLabel("Signature...");
    b3 = new JButton("Check");
    b3.addActionListener(this);
    b1 = new JButton("Select");
    b1.addActionListener(this);
    b4 = new JButton("Select");
    b4.addActionListener(this);
    b2 = new JButton("Select");
    b2.addActionListener(this);
    tf1 = new JTextField();
    centerPanel.add(l1);
    centerPanel.add(b1);
    centerPanel.add(l3);
    centerPanel.add(b4);
    centerPanel.add(l2);
    centerPanel.add(b2);
    centerPanel.add(b3);
    centerPanel.add(tf1);

    this.add(centerPanel,"Center");

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLocationRelativeTo(null);
    setVisible(true);
}

@Override
public void actionPerformed(ActionEvent e) {
    JButton button = (JButton) e.getSource();

```

```

    if (button==b1) {getfile();b1.setText(dir_patch);
    }
    else if (button==b2) {getfile();b2.setText(dir_patch);
    }
    else if (button==b4) {getfile();b4.setText(dir_patch);
    }
    else if (button==b3) {
        try {
            getdigital();
        } catch (NoSuchAlgorithmException ex) {
            Logger.getLogger(CheckDS.class.getName()).log(Level.SEVERE, null, ex);
        } catch (UnsupportedEncodingException ex) {
            Logger.getLogger(CheckDS.class.getName()).log(Level.SEVERE, null, ex);
        } catch (FileNotFoundException ex) {
            Logger.getLogger(CheckDS.class.getName()).log(Level.SEVERE, null, ex);
        } catch (NoSuchPaddingException ex) {
            Logger.getLogger(CheckDS.class.getName()).log(Level.SEVERE, null, ex);
        } catch (InvalidKeyException ex) {
            Logger.getLogger(CheckDS.class.getName()).log(Level.SEVERE, null, ex);
        } catch (IllegalBlockSizeException ex) {
            Logger.getLogger(CheckDS.class.getName()).log(Level.SEVERE, null, ex);
        } catch (BadPaddingException ex) {
            Logger.getLogger(CheckDS.class.getName()).log(Level.SEVERE, null, ex);
        } catch (InvalidKeySpecException ex) {
            Logger.getLogger(CheckDS.class.getName()).log(Level.SEVERE, null, ex);
        } catch (IOException ex) {
            Logger.getLogger(CheckDS.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
}

```

```

public void getfile(){
    JFileChooser fileChooser = new JFileChooser();

```

```

fileChooser.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
int ret = fileChooser.showDialog(this, "Open");
String patch = null;
if (ret == JFileChooser.APPROVE_OPTION)
{
    patch = fileChooser.getSelectedFile().getAbsolutePath();
}

```

```

        dir_patch = patch;
        final File folder = new File(patch);

    }
}
    public void getdigital() throws NoSuchAlgorithmException,
    UnsupportedEncodingException, FileNotFoundException,
    NoSuchPaddingException, InvalidKeyException, IllegalBlockSizeException,
    BadPaddingException, InvalidKeySpecException, IOException {
        String s = "";
        Scanner in = null;
        try {
            in = new Scanner(new File(b1.getText()));
        } catch (FileNotFoundException ex) {
            Logger.getLogger(DigitalSignature.class.getName()).log(Level.SEVERE,
            null, ex);
        }
        while(in.hasNext())
            s += in.nextLine();
        MD5Hash md5 = new MD5Hash();
        String hesh = md5.getHash(s);

        String sign = "";
        Scanner on = null;
        try {
            on = new Scanner(new File(b4.getText()));
        } catch (FileNotFoundException ex) {

            Logger.getLogger(DigitalSignature.class.getName()).log(Level.SEVERE, null,
            ex);
        }
        while(on.hasNext())
            sign += on.nextLine();
        RSAencrypt rsa = new RSAencrypt();
        String plainhash = rsa.RSAdecrypt(sign, b2.getText());
        if (hesh.equals(plainhash)) {tf1.setText("Successfully");
        tf1.setForeground(Color.green);}
        else {tf1.setText("Error");
        tf1.setForeground(Color.red);}
    }
}

```

```

        System.out.println(hesh);
        System.out.println(plainhash);
    }
}

```

Get Digital Signature frame

```

package cryptprotocols;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.math.BigInteger;
import java.security.*;
import java.security.spec.*;
import java.util.Scanner;
import java.util.logging.*;
import javax.crypto.*;
import javax.swing.*;
/*
 * @author Mumin
 */

public class DigitalSignature extends JFrame implements ActionListener{
    public String dir_patch = "";
    JLabel l1,l2,l3;
    JTextField tf1,tf2,tf3;
    JButton b1,b2,b3,b4;
    public DigitalSignature(){
        setLayout(new BorderLayout());
        setBackground(Color.WHITE);
        setSize(400, 175);

        Panel centerPanel = new Panel();
        centerPanel.setLayout(new GridLayout(5,2));
        l1 = new JLabel("Document for verification...");
        l2 = new JLabel("Private key...");
        l3 = new JLabel("Generate keys...");
        tf1 = new JTextField("Private key...");
        tf2 = new JTextField("Public key...");
        tf3 = new JTextField("Signature...");
        b1 = new JButton("Select");

```

```

b2 = new JButton("Select");
b3 = new JButton("Generate");
b4 = new JButton("GetDS");
b1.addActionListener(this);
b2.addActionListener(this);
b3.addActionListener(this);
b4.addActionListener(this);
centerPanel.add(l1);
centerPanel.add(b1);
centerPanel.add(l2);
centerPanel.add(b2);
centerPanel.add(l3);
centerPanel.add(b3);
centerPanel.add(tf1);
centerPanel.add(tf2);
centerPanel.add(tf3);
centerPanel.add(b4);
this.add(centerPanel, "Center");

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setLocationRelativeTo(null);
setVisible(true);

```

```

}

```

```

@Override

```

```

public void actionPerformed(ActionEvent e) {
    JButton button = (JButton) e.getSource();
    if (button==b1) {getFile();b1.setText(dir_patch);
    }
    else if (button==b2) {getFile();b2.setText(dir_patch);
    }
    else if (button==b3) {try {
        generateKeys(tf1.getText(), tf2.getText());
    } catch (NoSuchAlgorithmException ex) {

```

```

        Logger.getLogger(DigitalSignature.class.getName()).log(Level.SEVERE, null,
ex);

```

```

    } catch (InvalidKeySpecException ex) {

```

```
Logger.getLogger(DigitalSignature.class.getName()).log(Level.SEVERE, null,
ex);
    } catch (FileNotFoundException ex) {
```

```
Logger.getLogger(DigitalSignature.class.getName()).log(Level.SEVERE, null,
ex);
    } catch (IOException ex) {
```

```
Logger.getLogger(DigitalSignature.class.getName()).log(Level.SEVERE, null,
ex);
    }
}
else if (button==b4) {
    try {
        getdigital();
    } catch (NoSuchAlgorithmException ex) {
```

```
Logger.getLogger(DigitalSignature.class.getName()).log(Level.SEVERE, null,
ex);
    } catch (UnsupportedEncodingException ex) {
```

```
Logger.getLogger(DigitalSignature.class.getName()).log(Level.SEVERE, null,
ex);
    } catch (FileNotFoundException ex) {
```

```
Logger.getLogger(DigitalSignature.class.getName()).log(Level.SEVERE, null,
ex);
    } catch (NoSuchPaddingException ex) {
```

```
Logger.getLogger(DigitalSignature.class.getName()).log(Level.SEVERE, null,
ex);
    } catch (InvalidKeyException ex) {
```

```
Logger.getLogger(DigitalSignature.class.getName()).log(Level.SEVERE, null,
ex);
    } catch (IllegalBlockSizeException ex) {
```

```

Logger.getLogger(DigitalSignature.class.getName()).log(Level.SEVERE, null,
ex);
        } catch (BadPaddingException ex) {

Logger.getLogger(DigitalSignature.class.getName()).log(Level.SEVERE, null,
ex);
        } catch (InvalidKeySpecException ex) {

Logger.getLogger(DigitalSignature.class.getName()).log(Level.SEVERE, null,
ex);
        } catch (IOException ex) {

Logger.getLogger(DigitalSignature.class.getName()).log(Level.SEVERE, null,
ex);
        }
    }
}
public void getfile(){
    JFileChooser fileChooser = new JFileChooser();

fileChooser.setFileSelectionMode(JFileChooser.FILES_AND_DIRECTORIES);
    int ret = fileChooser.showDialog(this, "Open");
    String patch = null;
    if (ret == JFileChooser.APPROVE_OPTION)
    {
        patch = fileChooser.getSelectedFile().getAbsolutePath();
        dir_patch = patch;
        final File folder = new File(patch);
    }
}
    public void generateKeys(String publicFile,String privateFile) throws
NoSuchAlgorithmException, InvalidKeySpecException, FileNotFoundException,
IOException{
        KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("RSA");
        KeyPair keyPair = keyPairGen.generateKeyPair();
        PrivateKey privateKey = keyPair.getPrivate();
        PublicKey publicKey = keyPair.getPublic();

```



```
KeyFactory keyFactory = KeyFactory.getInstance("RSA");
RSAPrivateKeySpec rSAPrivateKey = keyFactory.getKeySpec(privateKey,
RSAPrivateKeySpec.class);
```

```
BigInteger pr_m = rSAPrivateKey.getModulus();
BigInteger pr_x = rSAPrivateKey.getPrivateExponent();
```

```
RSAPublicKeySpec rsaPublicKey = keyFactory.getKeySpec(publicKey,
RSAPublicKeySpec.class);
```

```
BigInteger pub_m = rsaPublicKey.getModulus();
BigInteger pub_x = rsaPublicKey.getPublicExponent();
```

```
String pubf = "keys/"+tf1.getText()+".txt";
String prf = "keys/"+tf2.getText()+".txt";
```

```
File publicKeyFile = new File(pubf);
File privateKeyFile = new File(prf);
```

```
FileOutputStream f1 = null;
FileOutputStream f2 = null;
```

```
f1 = new FileOutputStream(publicKeyFile);
f2 = new FileOutputStream(privateKeyFile);
```

```
f1.write(pub_m.toString().getBytes());
f1.write(" ".getBytes());
f1.write(pub_x.toString().getBytes());
```

```
f2.write(pr_m.toString().getBytes());
f2.write(" ".getBytes());
f2.write(pr_x.toString().getBytes());
```

```
tf1.setText(pubf);
tf2.setText(prf);
}
```

```
public void getdigital() throws NoSuchAlgorithmException,
UnsupportedEncodingException, FileNotFoundException,
```

```
NoSuchPaddingException, InvalidKeyException, IllegalBlockSizeException,  
BadPaddingException, InvalidKeySpecException, IOException {
```

```
    String s = "";  
    Scanner in = null;  
    try {  
        in = new Scanner(new File(b1.getText()));  
    } catch (FileNotFoundException ex) {
```

```
        Logger.getLogger(DigitalSignature.class.getName()).log(Level.SEVERE, null,  
ex);
```

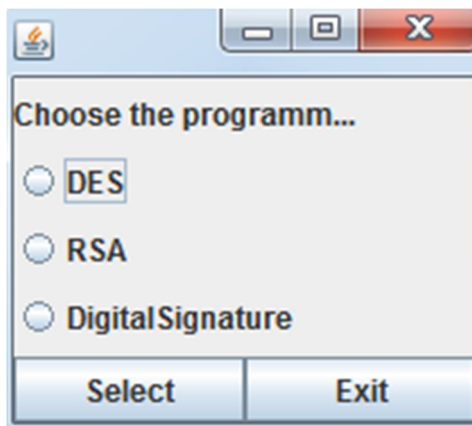
```
    }  
    while(in.hasNext())  
s += in.nextLine();  
MD5Hash md5 = new MD5Hash();  
String hashstr=md5.getHash(s);  
  
RSAencrypt rsa = new RSAencrypt();  
String result = rsa.RSAencrypt(hashstr, b2.getText());
```

```
String sign = "signatures/"+tf3.getText()+".txt";  
File file = new File(sign);  
FileOutputStream signfile = null;  
signfile = new FileOutputStream(file);  
signfile.write(result.getBytes());  
tf3.setText("Successfully");  
tf3.setForeground(Color.green);
```

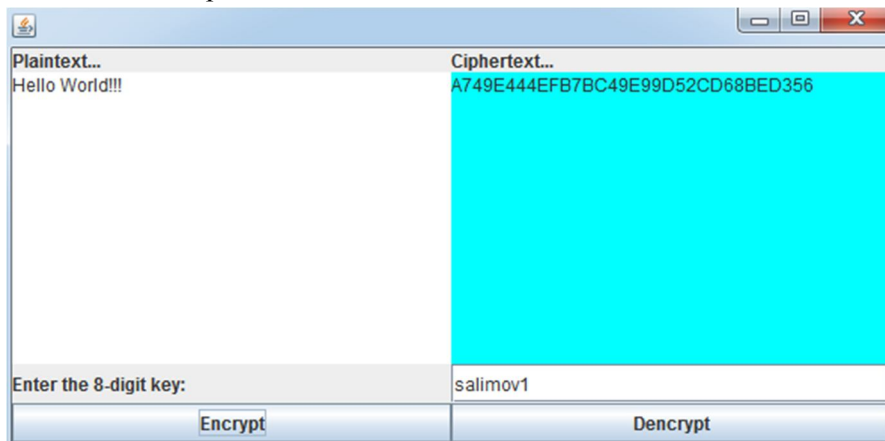
```
    }  
}
```

Program results

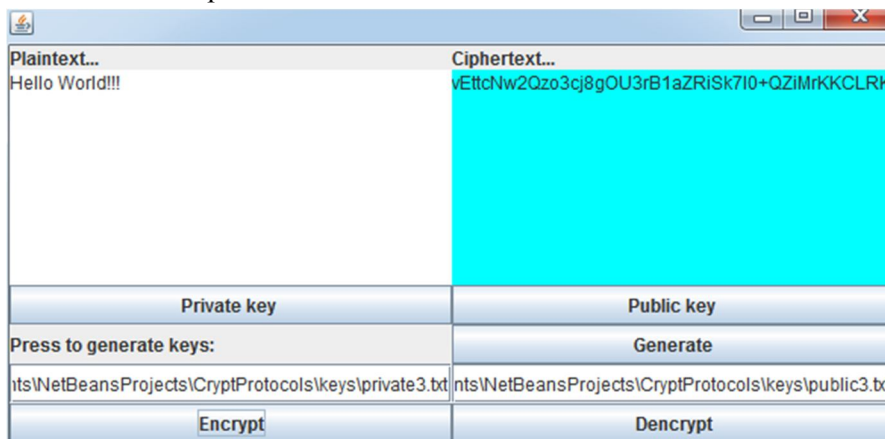
Main frame



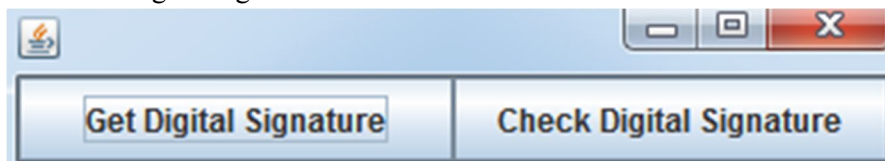
1. DES cipher



2. RSA cipher



3. Digital Signature



3.1. Get DigitalSignature

Document for verification...	Select
Private key...	Select
Generate keys...	Generate
Private key...	Public key...
Signature...	GetDS

3.2. Check Digital Signature

Document for cheking...	Select
Signature...	Select
Public key...	Select
Check	