OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

INFORMATIK UND
MATHEMATIK

Secure Programming

Asymmetric Encryption

Prof. Dr. Christoph Skornia
christoph.skornia@oth-regensburg.de

Fundamentals of encryption

Definition: A cryptographic cipher is defined as a

$$(\mathcal{M}, \mathcal{C}, \mathcal{K}_E, \mathcal{K}_D, E, D)$$

with

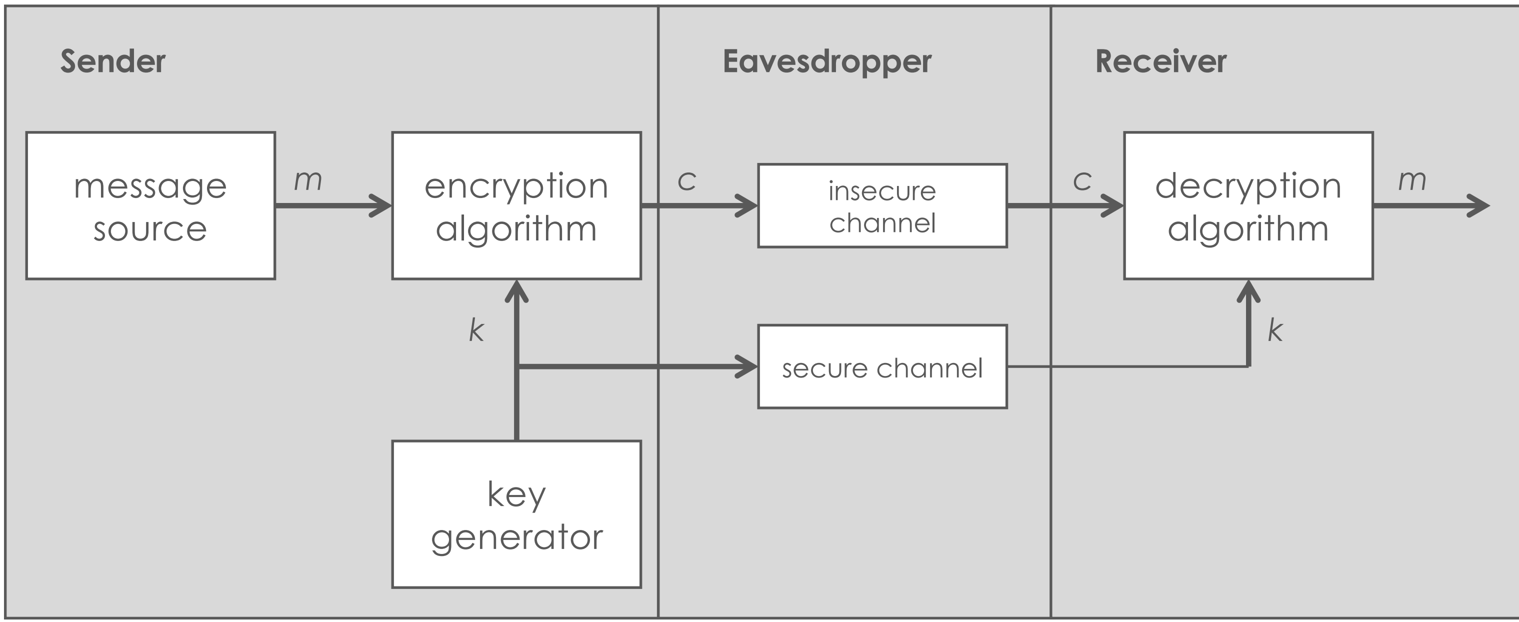
1. $\mathcal{M}$: Message Space
2. $\mathcal{C}$: Cryptogram Space
3. $\mathcal{K}_E$: Encryption-Keyspace
4. $\mathcal{K}_D$: Decryption-Keyspace and a function $f : \mathcal{K}_E \longrightarrow \mathcal{K}_D$
5. The injective encryption function $E : \mathcal{M} \times \mathcal{K}_E \longrightarrow \mathcal{C}$
6. The decryption function $C : \mathcal{C} \times \mathcal{K}_D \longrightarrow \mathcal{M}$

in the way that

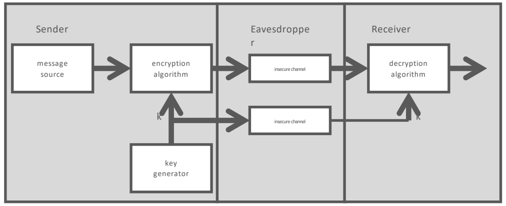
$$D(E(M, K_E), f(K_E)) = M \quad \forall M \in \mathcal{M}, K_E \in \mathbb{D}(f)$$

Fundamentals of encryption

Diagram of a private-key encryption



Fundamentals of encryption



Key distribution is a constant source or insecurity

So what to do?

Solution:

1. every person p being part of the communication creates a pair of keys $K_E(p_i)$ and $K_D(p_i)$ which can be used for encryption and decrypton in an assymetric cipher.
2. everyone shares its personal $K_E(p)$ with everone else
3. if p_1 want to send an encrypted message to p_2 the encryption will be done with $K_E(p_2)$, and therefore decryption can only be done with $K_D(p_2)$ and therefore only by p_2

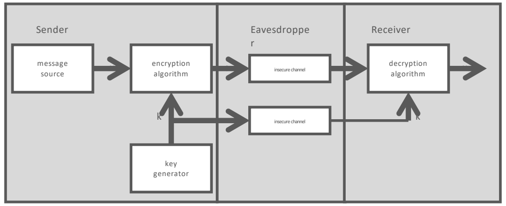
This way of doing assymetric encryption is called Public Key Cryptograpy and $K_E(p_i)$ is also called p_is public key and $K_D(p_i)$ p_is secret key.

OSTBAYERISCHE TECHNISCHE HOCHSCHULE REGENSBURG
INFORMATIK UND MATHEMATIK

Fundamentals of encryption

Requirements for Public-Key-Cryptography:

- computing the secret key from the public key must be intractable for any public key
- Recover the plaintext message from the from encrypted message and the public key must be intractable for any encrypted message and any public key



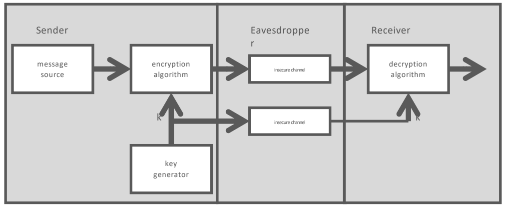
Fundamentals of encryption

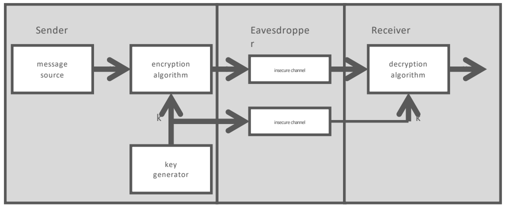
Example (RSA):

1. choose two prime numbers p and q and compute $N = p \cdot q$ and $\phi(N) = (p-1)(q-1)$
2. choose an integer $1 < e < \phi(N)$ wheras e and $\phi(N)$ are coprime and determine $d := e^{-1} \text{mod}(\phi(N))$
3. $K_E = (N, e)$ and $K_D = (N, d)$
4. convert the message into an integer $0 \leq m < N$
5. Encryption: $c = m^e \text{mod}(N)$
6. Decryption: $m = c^d \text{mod}(N)$

Be happy!!!!

[Simulation]



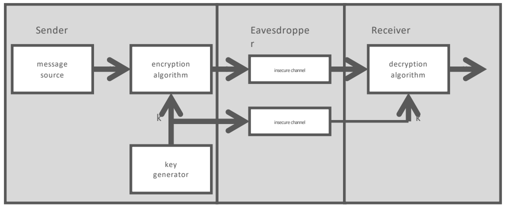


Security Considerations:

Two types of attacks:

- ❑ derive K_D from K_E(public key)
  - can be derived if $\phi(N)$ can be calculated
  - can be calculated if N can be factorized
    $\Longrightarrow$ RSA is as secure as factorization
- ❑ derive m from c (without knowing K_D)
  - calculate m if c and K_E is known
    $\longrightarrow$ known as **RSA-Problem**
  - maybe related to factorization but no proof found so far

Further attacks:

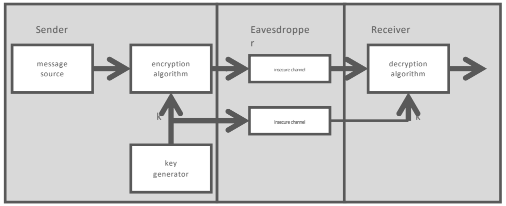


- ❑ known plaintext attacks
  - derive d from m N, and c
  - related to discrete logarithm
- ❑ timing attacks
  - implementation which takes more cpu-time for larger d
  - mearuring performance helps to reduce effective keyspace
- ❑ sidechannel attacks (e.g. power attacks)
  - measure HW-properties to reduce effective keyspace

https://www.openssl.org/news/secadv/20030317.txt
https://www.openssl.org/news/secadv/20181030.txt

Fundamentals of encryption

Sounds quite good but what about integrity and authentication?



Integrity could be added easily with hashes.

Solution for authentication:

1. find a pair of functions $S : \mathcal{M} \times \mathcal{K}_D \longrightarrow \Sigma^n$ and $V : \mathcal{C} \times \Sigma^n \times \mathcal{K}_E \longrightarrow \{1, 0\}$ with the property:

$$V(C, S(M, K_D), K_E) = \begin{cases} 1 & \text{if } K_D \text{ is the corresponding secret key to } K_E \\ 0 & \text{if not} \end{cases}$$

2. the sender p_1 adds $S(M, K_D(p_1))$ to the encrypted message C
3. the receiverp_2 proofs that the sender knows the private key $K_D(p_1)$
4. if only p_1 knows $K_D(p_1)$ then the sender must be p_1

This procedure is called signing a message with a Ditigal Signature

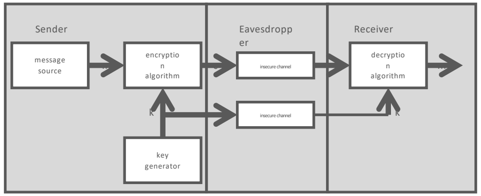
Fundamentals of encryption

How to find such a signing function?

Solution:

If we assume we have a assymetric cipher, where we can use the private key also for encryption.

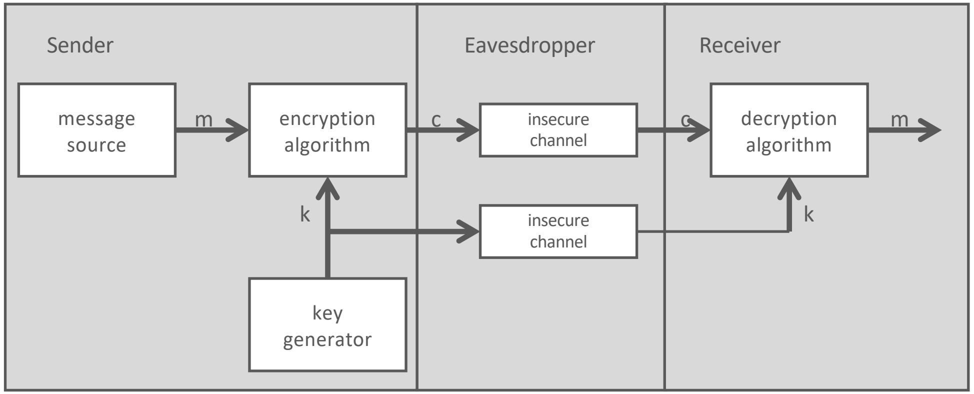
1. sender p_1 calculates a hash $H_1(M)$ of the plaintext message
2. p_1 encrypts $H_1(M)$ with the private key $K_D(p_1)$
3. p_1 encrypts the message with $K_E(p_2)$
4. p_1 sends the encrypted message and the encrypted hast to the receiver p_2
5. p_2 decrypts the message with $K_D(p_2$ calculates its own hash $H_2(M$
6. p_2 decrypts the transmitted hash with $K_E(p_1)$ and compares the result with his own hashresult.
7. if they two hashes match then p_2 knows that the message was encrypted by p_1 and not changed since p_1 created the hash



Be happy!!!

Fundamentals of encryption

All we need is such a cipher and we have it already with RSA:



Remember:

$d := e^{-1} \text{mod}(\phi(N))$

$\Rightarrow$ e and d are just inverse

$\Rightarrow$ no matter which one to use for encryption or decryption.

$\Rightarrow$ Bingo!!!

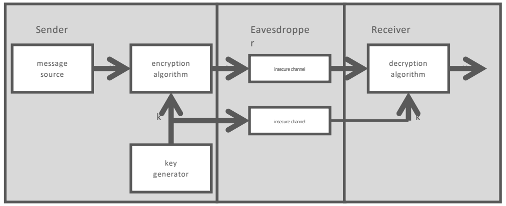
Fundamentals of encryption

Problem: What if the private key is stolen? $\Rightarrow$ Message can be decrypted...

Diffie-Hellman Keyexchange: Let p be prime and g be a primitive root of p

Secret	Public	Calculates	Sends	Calculates	Public	Secret
a	p, g		$p, g \rightarrow$			b
a	p, g, A	$g^a \mathrm{mod} p = A$	$A \rightarrow$		p, g	b
a	p, g, A		$\leftarrow B$	$g^b \mathrm{mod} p = B$	p, g, A, B	b
$a, \boldsymbol{s}$	p, g, A, B	$B^a \mathrm{mod} p = s$		$A^b \mathrm{mod} p = s$	p, g, A, B	$b, \boldsymbol{s}$

- Now we can share keys secretly
- Combining this with authentication through RSA (or similar) delivers pretty good privacy...



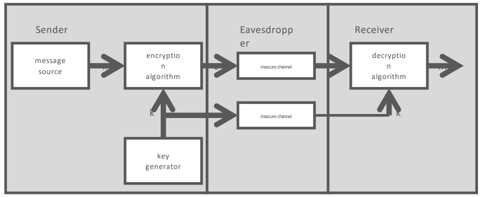
Fundamentals of encryption

Note:

- In reality the plaintext message is padded with random data (like a salt) to avoid know plaintext attacks

- widely used key ciphers: RSA, DSA, El Gamal, ECDSA

- PK-encryption is 1000 times slower than symmectric-encryption
  ⇒ most of the time PK-encryption is used to encrypt symmetric key to transfer it over an insecure channel.
  ⇒ no secure channel needed and high speed!!



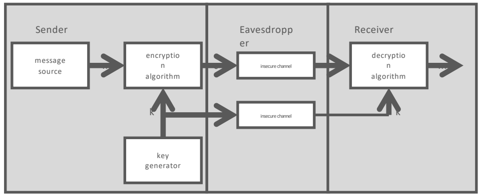
Fundamentals of encryption

So what do we need now for programming?

Choose algorithm and key length.

Desired security level	Symmetric length	"Regular" public key lengths	Elliptic curve sizes
Acceptable (probably secure 5 years out, perhaps 10)	80 bits	1024 bits	160 bits
Good (may even last forever)	128 bits	2048 bits	224 bits
Paranoid	192 bits	4096 bits	384 bits
Very paranoid	256 bits	8192 bits	512 bits

For our examples lets use: RSA with 4096bits



Fundamentals of encryption

Generate keypair!

```
RSA *RSA_generate_key(int bits, unsigned long exp, void (*cb)(int, int, void), void *cb_arg);
```

bits: Size of the key to be generated, in bits. This must be a multiple of 16, and at a bare minimum it should be at least 1,024. 2,048 is a common value, and 4,096 is used occasionally. The more bits in the number, the more secure and the slower operations will be. We recommend 2,048 bits for general-purpose use.

exp: Fixed exponent to be used with the key pair. This value is typically 3, 17, or 65,537, and it can vary depending on the exact context in which you're using RSA. For example, public key certificates encode the public exponent within them, and it is almost universally one of these three values. These numbers are common because it's fast to multiply other numbers with these numbers, particularly in hardware. This number is stored in the RSA object, and it is used for both encryption and decryption operations.

cb: Callback function; when called, it allows for monitoring the progress of generating a prime.

cb_arg: Application-specific argument that is passed directly to the callback function, if one is specified.

OSTBAYERISCHE TECHNISCHE HOCHSCHULE REGENSBURG
INFORMATIK UND MATHEMATIK

Fundamentals of encryption

Encrypt/Decrypt!

```
int RSA_public_encrypt(int l, unsigned char *pt, unsigned char *ct, RSA *r, int p);
```

```
int RSA_public_decrypt(int l, unsigned char *pt, unsigned char *ct, RSA *r, int p);
```

```
int RSA_private_decrypt(int l, unsigned char *ct, unsigned char *pt, RSA *r, int p);
```

```
int RSA_private_encrypt(int l, unsigned char *pt, unsigned char *ct, RSA *r, int p);
```

l: Length of the plaintext to be encrypted.

pt: Buffer that contains the plaintext data to be encrypted.

ct: Buffer into which the resulting ciphertext data will be placed. The size of the buffer must be equal to the size in bytes of the public modulus. This value can be obtained by passing the RSA object to RSA_size( ).

r: RSA object containing the needed key data

p: Type of padding to use.

RSA_PKCS1_PADDING
EME-OAEP
RSA_SSLV23_PADDING
RSA_NO_PADDING

Note: l must be less than RSA_size(rsa) - 11 for the PKCS #1 v1.5 based padding modes, less than RSA_size(rsa) - 41 for RSA_PKCS1_OAEP_PADDING and exactly RSA_size(rsa) for RSA_NO_PADDING.

Fundamentals of encryption

```
int RSA_sign(int md_type, unsigned char *dgst, unsigned int dlen,
             unsigned char *sig, unsigned int *siglen, RSA *r);
```

```
int RSA_verify(int type, const unsigned char *m, unsigned int m_len,
               unsigned char *sigbuf, unsigned int siglen, RSA *rsa);
```

md_type: OpenSSL-specific identifier for the hash function. Possible values are NID_sha1, NID_ripemd, or NID_md5. A fourth value, NID_md5_sha1, can be used to combine MD5 and SHA1 by hashing with both hash functions and concatenating the results. These four constants are defined in the header file openssl/objects.h.

dgst: Buffer containing the digest to be signed. The digest should have been generated by the algorithm specified by the md_type argument.

dlen: Length in bytes of the digest buffer. For MD5, the digest buffer should always be 16 bytes. For SHA1 and RIPEMD, it should always be 20 bytes. For the MD5 and SHA1 combination, it should always be 36 bytes.

sig: Buffer into which the generated signature will be placed.

siglen: The number of bytes written into the signature buffer will be placed in the integer pointed to by this argument. The number of bytes will always be the same size as the public modulus, which can be determined by calling RSA_size( ) with the RSA object that will be used to generate the signature.

r: RSA object to be used to generate the signature. The RSA object must contain the private key for signing.

thanks for your interest

to be continued