Cryptography

Outline

- Cryptographic hash function and HMAC
- Symmetric encryption
- Symmetric key and hash lengths
- Public-key signature
- Public-key encryption
- Diffie-Hellman key exchange
- Summary (notes about cryptography)

CRYPTOGRAPHIC HASH FUNCTION AND HMAC

Cryptographic hash
= message digest
= fingerprint

Legacy installer: putty-0.67-ins

Checksums for all the above files

MD5:	md5sums
SHA-1:	sha1sums
SHA-256:	sha256sums
SHA-512:	sha512sums

The latest development snapshot

Philip Zimmermann

Phil's Public Keys

For a copy of these keys you can import directly into PGP, click here.

Current DSS/Diffie-Hellman Key:

```
Key fingerprint:
055F C78F 1121 9349 2C4F  37AF C746 3639 B2D7 795E
```

Older DSS/Diffie-Hellman Key:

```
Key fingerprint:
17AF BAAF 2106 4E51 3F03  7E6E 63CB 691D FAEB D5FC
```

```
$ git log
commit 9036c57ab9275f0e42f63a391ed68044f8c590bc
Author: raghunfs
Date:   Fri Jul 1 07:44:23 2016 +0000

    Handling error codes

commit 4d057be278eedce4e2c0682604d5304c7d18fb5a
Author: ms88 <ms88>
Date:   Tue Jun 28 16:27:27 2016 +0300

    fix fast reconnect
```

BLOCKCHAIN info — Home | Charts | Stats | Markets | API | Wallet

Block #431985

Hashes	
Hash	00000000000000000030166f5cadfc133e2e7474d880f9050f6503ac5c19edf45
Previous Block	000000000000000004247a0e018c3810c660fded6d35591b8a41fe0507ef012b

Cryptographic hash function

input: any byte string

↓

Public pseudorandom function

↓

output: short n-bit string

"Hello!"

SHA-256

```
334d016f 755cd6dc
58c53a86 e183882f
8ec14f52 fb053458
87c8a5ed d42c87b7
```

256 bits = 32 bytes

- The algorithm is public, no keys or other secrets needed
- Examples: SHA-256, SHA-512, SHA3-256

`echo -n "Hello!" | openssl sha256`

Cryptographic hash: security requirements

- One-way = pre-image resistant: given only output, impossible to compute input, except by guessing
- Second-pre-image resistant: given one input, impossible to find a second input that produces the same output
- Collision-resistant: impossible to find *any* two inputs with the same output
  - Old hash functions with broken collision resistance: MD5, SHA-1

Hash function implementation

- Ideal hash function is a random, public function chosen from the set of all byte strings (of any length) to bit-strings of fixed-length (e.g. n=256 bits)
  – Also called "random oracle"
  – In practice, impossible to store and share such infinite-size functions
- Practical hash function is pseudorandom: deterministic algorithm, but output looks random
  – One-way, collision resistant
  – Efficient to compute for large inputs
  – Typically algorithm based on And, Xor, Rot, Add (mod 2^{32}) operations

Hash function applications

- Integrity check on stored files, software downloads, or any data – compute hash and compare with known correct value
- Unique, “self-certifying” identifier for any object, e.g. file, public key, Bitcoin block
- Key derivation and password storage, e.g. PBKDF2
- Signing: sign the hash of the message with RSA
- Message authentication with HMAC and a shared secret key

Hash collisions

- Research has found collisions in several standard hash functions
  - MD5, SHA-1
  - Applications should be designed for crypto agility i.e. easy upgrading of functions
- Where and why is collision resistance needed?
  (or is preimage and second-preimage resistance sufficient?)
  - File integrity check?
  - Software integrity check?
  - Digital signature on a contract?
  - MAC for end-to-end authentication?
  - Password storage?
  - Key derivation in Wi-Fi?
  - Bitcoin?
- Not all applications need collision resistance, but many do in subtle ways

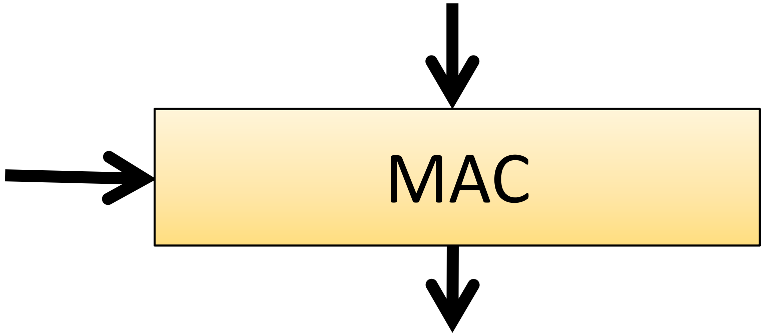
Message authentication code (MAC)

input: any byte string

"Hello!"

"EZ87k7QC7m
9c8t23pG37s
rrpU9y8KZHP
nmmHG427DdU
6G632"

Secret key K



HMAC-SHA256

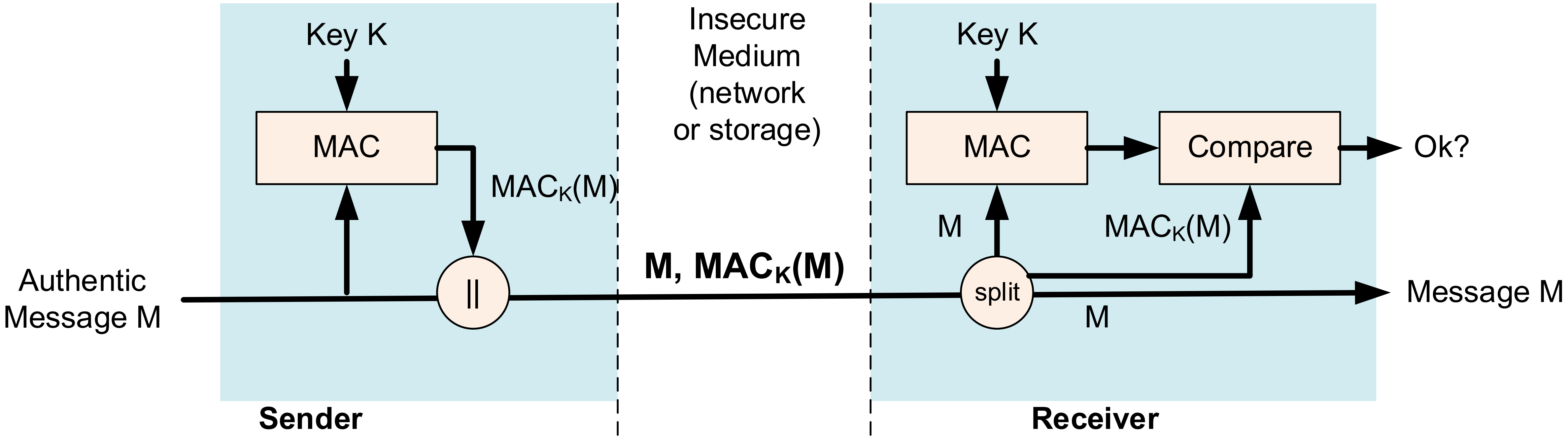
output: short n-bit string

3b8a41f7 b44336ae
ca058ae6 94a06be1
43f6d366 54d79abd
a5300240 47ea9746

- Secret key is needed to create and to check the MAC
- HMAC is a standard way to construct a MAC from a hash function, e.g. HMAC-SHA256

```
echo -n "Hello!" | openssl sha256 -hmac "EZ87k7QC7m9c8t23pG37srrpU9y8KZHPnmmHG427DdU6G632"
```

Message authentication with MAC



- Message authentication and integrity protection
- Endpoints share the secret key K (thus, it is symmetric cryptography)
- MAC is appended to the original message M

HMAC details

- HMAC is commonly used in standards:
  - Way of deriving MAC from a cryptographic hash function h

  $HMAC_K(M) = h((K \oplus opad) \mid h((K \oplus ipad) \parallel M))$

  - Hash function h is instantiated with SHA-1, MD5 etc. to produce HMAC-SHA-1, HMAC-MD5,…
  - $\oplus$ is XOR; | is concatenation of byte strings
  - ipad and opad are bit strings for padding the key to fixed length
  - Details: [RFC 2104][Bellare, Canetti, Krawczyk Crypto’96] *
- HMAC is theoretically stronger than simpler constructions, e.g. $h(M \mid K)$

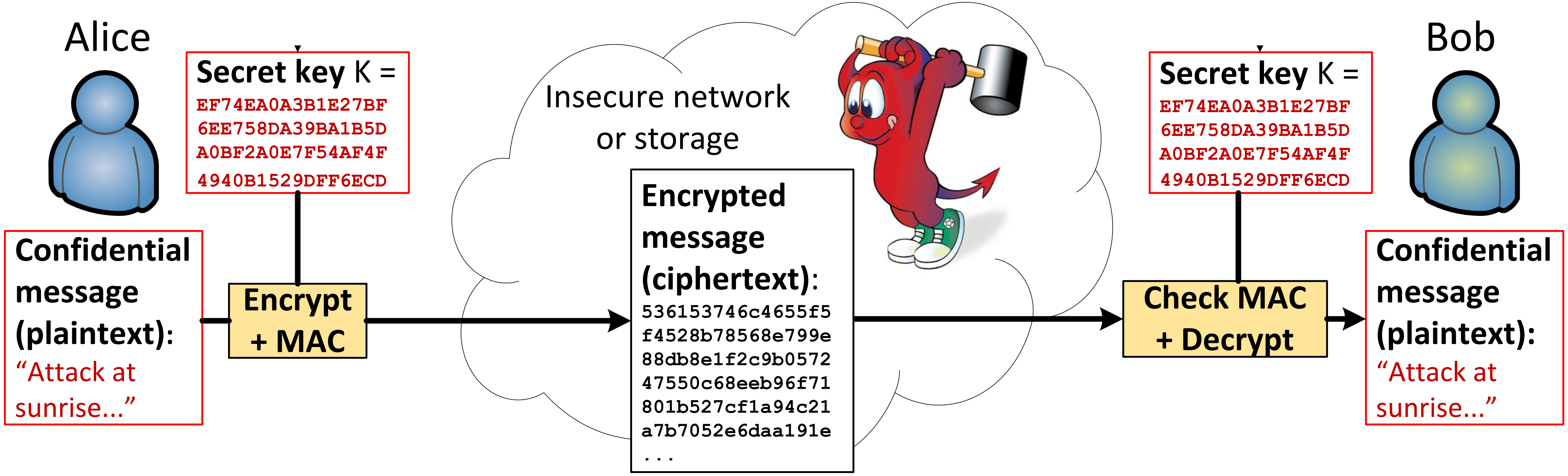
Hash and HMAC commands

```
# Compute the hash of a file
echo "Attack at sunrise!" > m.txt
sha256sum m.txt
openssl dgst -sha256 m.txt
# Append a LF to the file and see if the hash changes
echo >> m.txt
openssl dgst -sha256 m.txt

# Compute HMAC using hash of "abc123" (bad!) as the key
openssl dgst -sha256 -hmac abc123 m.txt
# Change the key slightly and see if the hash changes
openssl dgst -sha256 -hmac abc132 m.txt
```

SYMMETRIC ENCRYPTION

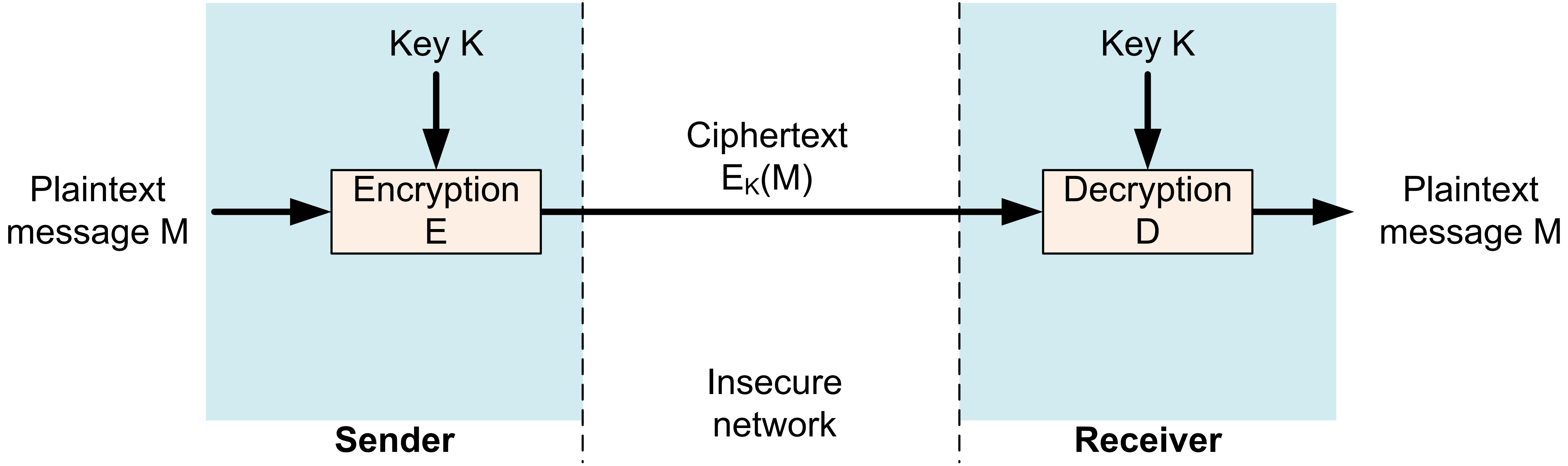
Symmetric encryption



- Message encryption based on symmetric cryptography, i.e. a shared secret key

[Daemon graphics: www.freebsd.org]

Symmetric encryption

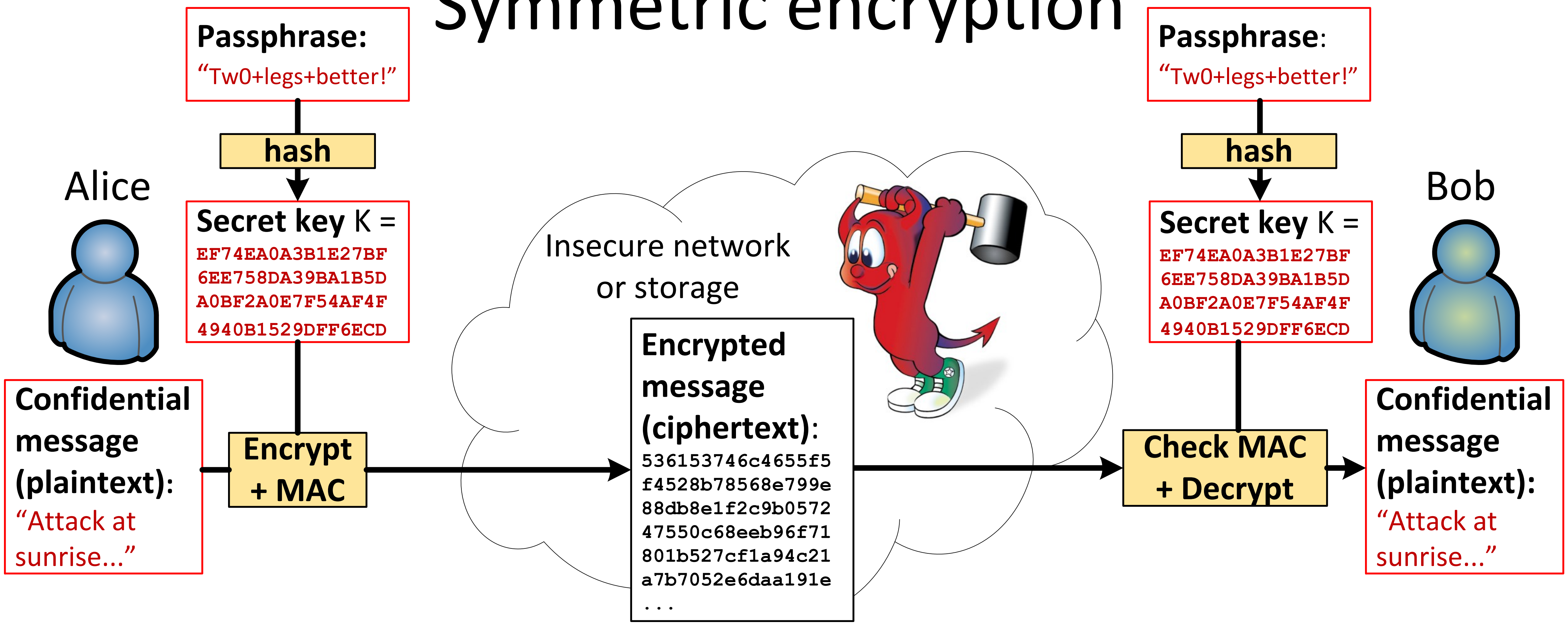


- Message encryption based on symmetric cryptography, i.e. a shared secret key

Symmetric encryption

- Kerckhoff's principle: the encryption and decryption algorithms are public algorithm; only the key is secret
- Encrypted message content looks like random bits – unless you know the key
- The key must be shared over a secure out-of-band channel
  - a 128…256-bit random number
  - sometimes computed from a passphrase with a cryptographic hash function (should use PBKDF2 to make cracking slower)

Symmetric encryption



- Message encryption based on symmetric cryptography, i.e. a shared secret key

[Daemon graphics: www.freebsd.org]

Block cipher and cipher mode

- Block cipher is the basic construction block for encryption
  - Encryption of a fixed-length block, typically 128 bits
  - Examples: AES, 3DES
- Cipher mode uses the block cipher as building block for encrypting messages of any length
  - Padding of the message to full blocks
  - Initialization vector, so that the same plaintext always produces a different ciphertext (called salt in OpenSSL commands)
  - Example: cipher-block chaining (CBC)

Symmetric encryption with OpenSSL

```
# Create a plaintext message (length multiple of 128 bits).
echo "Secret meeting in the usual place at 10 am xxxx" > m.txt
hexdump -C m.txt

# Encrypt with block cipher.
openssl enc -aes-256-cbc -nosalt -nopad -k abc123 -in m.txt -out m.enc
cat m.enc
hexdump -C m.enc
# Note how random the ciphertext looks. Then, decrypt and compare.
openssl enc -d -aes-256-cbc -nosalt -nopad -k abc123 -in m.enc -out r.txt
hexdump -C r.txt
# Try also decrypting with a different key.
# Edit the ciphertext slightly and decrypt again. The plaintext may change only partly.

# Normally, encryption uses salt (or IV) and padding: The salt is random, not secret, and stored with the ciphertext. The
message is padded to full 128-bit blocks.
echo "Secret meeting in the usual place at 10 am." > m.txt
hexdump -C m.txt
openssl enc -aes-256-cbc -k abc123 -in m.txt -out m.enc
hexdump -C m.enc
openssl enc -d -aes-256-cbc -k abc123 -in m.enc -out r.txt
hexdump -C r.txt
# Edit one byte of the ciphertext and decrypt again.

# OpenSSL computes the key (and IV) from with PBKDF2 from the passphrase and salt.
# If we encrypt the same message again, thanks to the salt, the ciphertext looks different.
hexdump -C m.enc
openssl enc -aes-256-cbc -k abc123 -in m.txt -out m.enc
hexdump -C m.enc

# Encrypted files are binary. To send over email or http, they are usually base64 encoded.
openssl enc -aes-256-cbc -base64 -k abc123 -in m.txt -out m.enc
cat m.enc
```

Encryption and message integrity

- Encryption alone protects secrets, not integrity !
  - Attacker can usually modify the secret message
  - Receiver of the modified secret message usually leaks some information, e.g. error in message

➔ Always combine encryption with integrity protection !

  - Encrypt-then-MAC: encrypt with block cipher e.g. in CBC mode, then compute and append a MAC
  - Authenticated encryption modes do encryption and integrity in one pass, e.g. AES-GCM

If in doubt, use Authenticated encryption with associated data (AEAD)

SYMMETRIC KEY AND HASH LENGTHS

Key length (1)

- Shared key of ≥ 128 bits is strong, < 80 bits is weak
  - To resist brute-force guessing, the secret key must be random with (almost) even probability distribution
  - Quantum cryptoanalysis may require keys of 256 bits in the future
  - Q: Why is a secret key of 1000 bits on 1 MB not better than 256?

Number of atoms in the earth is less than $10^{50} \approx 2^{166}$.
Age of the universe $4.3 \cdot 10^{17} \approx 2^{59}$ seconds $\approx 2^{89}$ nanoseconds.
$2^{166} \cdot 2^{89} \leq 2^{256}$.
→ 256-bit keys definitely cannot be brute-forced

Key length (2)

- Brute-force attacks are easy to parallelize; thus, cost should never be measured in time but in money (EUR, USD, CPU days)
  - 1 CPU day = $1 on high-end PC, less on cloud infrastructure
  - Q: If NSA has a billion-dollar computer and can break DES encryption keys in 1 second, how much does it cost for you to break them on Amazon EC2?

- Strength of a key derived from passphrase?

  K = SHA-256(“verYsekReTT123pasSfraZe”)

  - Dictionary attack to guess human-invented passphrases is possible, while brute-forcing a random 128 or 256-bit key is not

Hash length and birthday paradox

- How long hash values? Answer: 256..512 bits
- One-wayness and second preimage resistance require has length of 128..256 bits. Why?
  - Attacker tries different inputs to match a known hash value. Impossible to perform 2^{128} hash computations
- Collision resistance requires almost twice that length. Why?
- Birthday attack: *store* computed hash values and find a match between *any two* of them

Hash length and birthday paradox (2)

- Rule of thumb: When randomly sampling a set of M values, collisions appear after $M^{1/2}$ (square root of M) samples

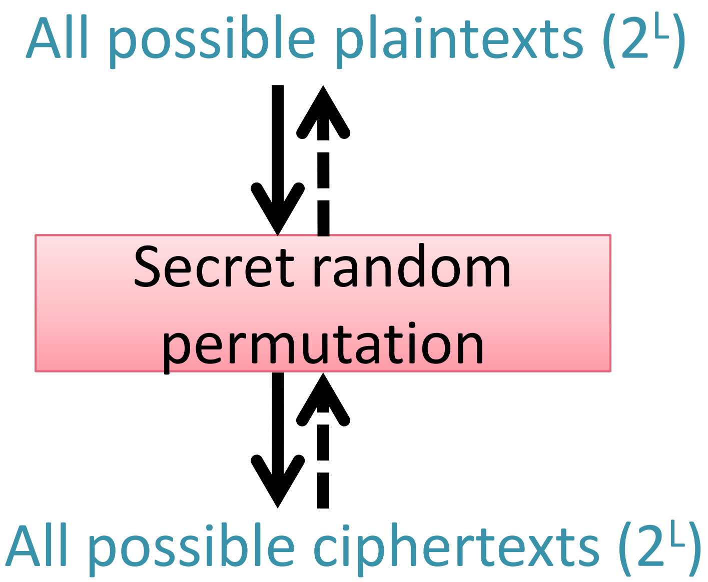
  (More precisely: for large M, the collision probability is 50% at $(2 \cdot \ln 2 \cdot M)^{1/2} \approx 1.18 \cdot M^{1/2}$ samples.)

- Same rule in different words:
  - When randomly sampling a set of 2^N values, collisions appear after $2^{N/2}$ samples
  - If attacker can compute and store 2^N hash values, it can find collisions for hash values of length $2 \cdot N$ bits
  - If an N-bit hash value is safe against brute-force reversing, nearly $2 \cdot N$ bits are needed to avoid collisions with birthday attack ("nearly" because brute-force reversing requires only CPU but the birthday attack requires also storage)

HOW DOES ENCRYPTION WORK? – BLOCK CIPHERS

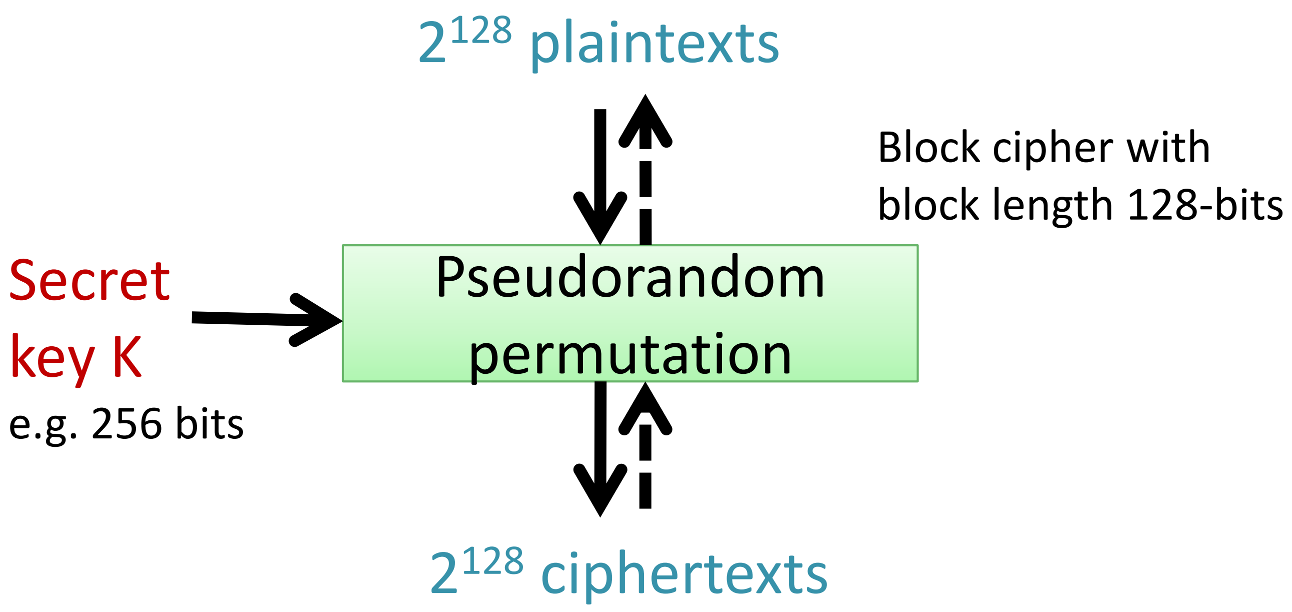
Please read this section for a rough idea of how a block cipher works. More details in a cryptography course

Ideal encryption: random permutation



- Messages = bit strings with some maximum length L
- Ideal encryption would be a random 1-to-1 function i.e. permutation of the set of all possible messages to itself
- Decryption is the reverse function
- Like an old-fashioned military code book, but *much* larger
- Impossible to store and share: table with 2^L rows

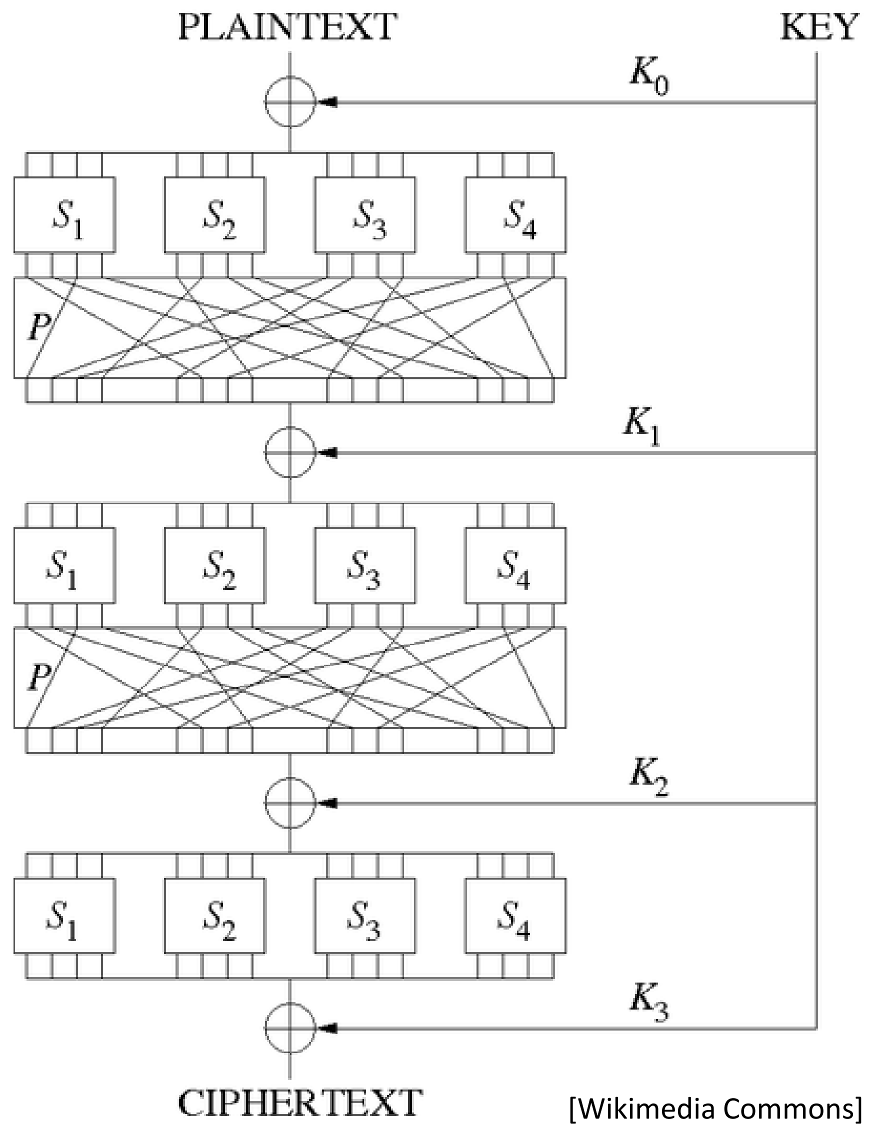
Real encryption: pseudorandom permutation



- Block cipher: string length fixed usually to L=128 bits
  - Pseudorandom permutation that depends on a secret key of 128..256 bits
  - Number of different permutations is 2^{256}, large but far less than $(2^L)!$
- Pseudorandom = indistinguishable from random unless you know the algorithm and key
- Kerckhoff’s principle: public algorithm, secret key

Substitution-permutation network

- One way to implement a key-dependent pseudorandom permutation
- Substitution-permutation network:
  - S-box = substitution is a small (random) 1-to-1 function for a small block, e.g. $2^4 \dots 2^{16}$ values
  - P-box = bit-permutation mixes bits between the small blocks
  - Repeat for many rounds, e.g. 8…100
  - Mix key bits with data in each round
  - Decryption is the reverse
- Cryptanalysis tries to detect minute differences between this and a true random permutation



[Wikimedia Commons]

Cipher design

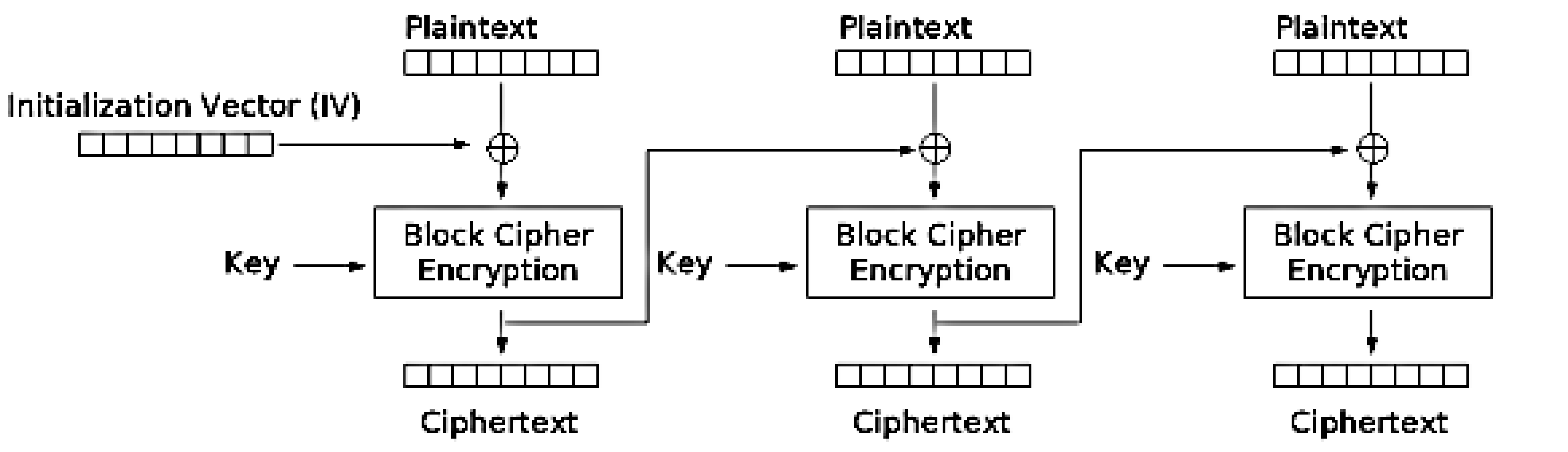
- It is not difficult to make strong block cipher: long key, large S-boxes, many many rounds
- Good bock ciphers are not only strong
  - fast to compute in software
  - require little memory
  - cheap to implement in hardware
  - optimized for both throughput and latency
  - use a short (e.g. 128-bit) key, which is expanded to the round keys, but still allow fast key changes
  - no unexplained features that could be a backdoor
  - implementation is resistant to side-channel attacks
  - etc.
- The difficulty is in finding a balance between performance and security

AES

- Advance Encryption Standard (AES)
  - Standardized by NIST in 2001
  - 128-bit block cipher
  - 128, 192 or 256-bit key
  - 10, 12 or 14 rounds
- AES round:
  - SubBytes: 8-byte S-box, not really random, based on finite-field arithmetic, multiplication in GF(2^8)
  - ShiftRows and MixColumn: reversible linear combination of S-box outputs (mixing effect similar to P-box)
  - AddRoundKey: XOR bits from expanded key with data
- Key schedule: expands key to round keys

Cipher mode example

- Block-cipher mode, e.g. cipher-block chaining (CBC), is used for encrypting longer messages



Cipher Block Chaining (CBC) mode encryption [Wikimedia Commons]

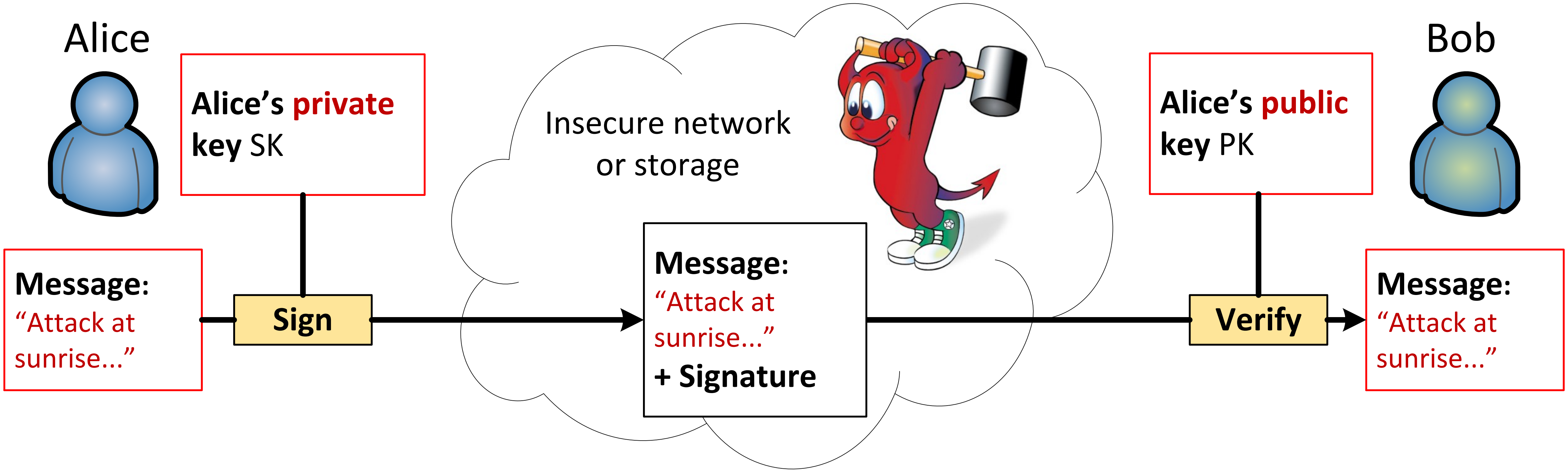
- Initialization vector (IV) makes ciphertexts different even if the message repeats. It may be a non-repeating counter or a random number that is also sent to the receiver. IV is not secret
- The message is padded to fill full blocks of the block cipher

Common ciphers and modes

- Block ciphers:
  - DES — old standard, 56-bit keys now too short, 64-bit block
  - 3DES in EDE mode: DESK3(DES-1K2(DESK1(M))), 3x56 key bits
  - AES — at least 128-bit keys, 128-bit block
- Block-cipher modes
  - E.g. electronic code book (ECB), cipher-block chaining (CBC)
- Stream ciphers:
  - XOR plaintext and a keyed pseudorandom bit stream
  - RC4: simple and fast software implementation
- Most encryption modes are malleable: attacker can make controlled modifications to the plaintext
  - E.g. consider CBC mode or stream cipher
- Authenticated encryption modes combine encryption and integrity check

ASYMMETRIC CRYPTOGRAPHY: DIGITAL SIGNATURE

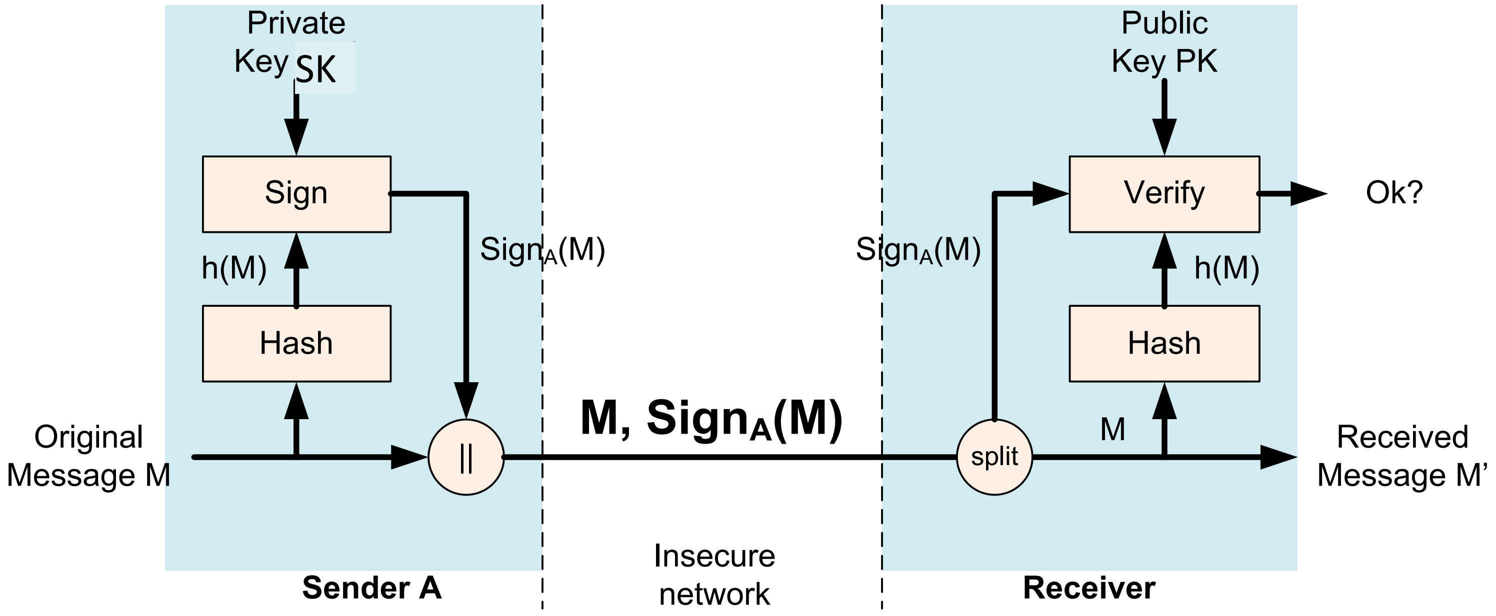
Digital signature



- Message authentication and integrity protection
- Asymmetric i.e. public-key cryptography
- Key pair with public and private parts

[Daemon graphics: www.freebsd.org]

Digital signature



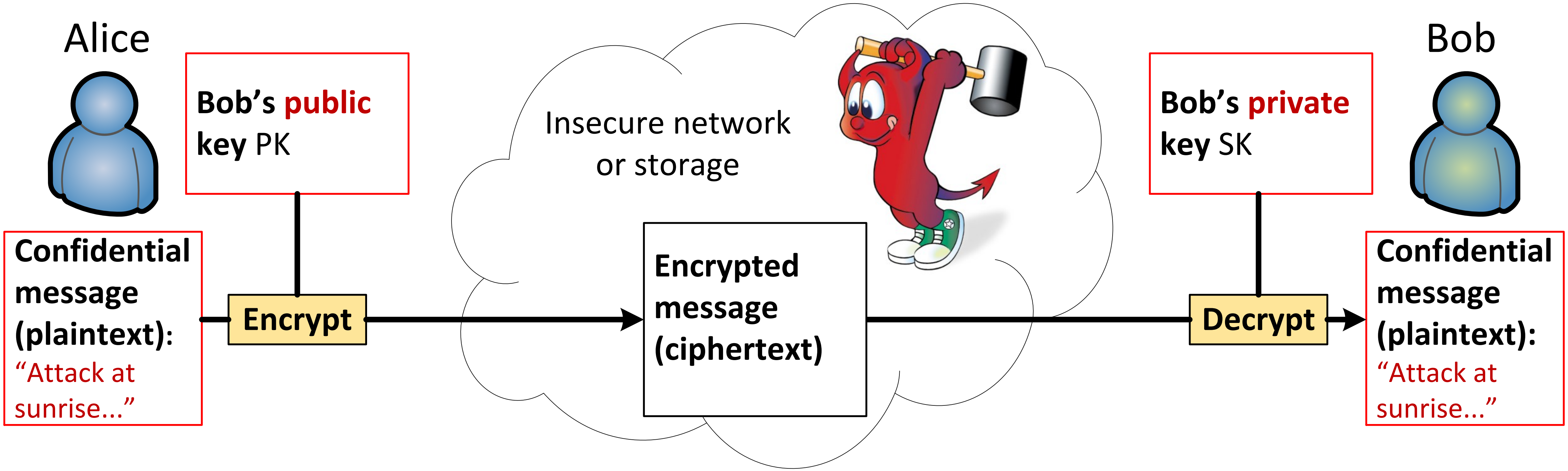
- Message authentication and integrity protection with public-key crypto
  - Verifier has a public key PK ; signer has the private key SK
  - Messages are first hashed and then signed
  - Examples: DSS, RSA + SHA-256, ECDSA

Digital signature issues

- Always follow strictly the standard when implementing signatures! There are many subtle points that can go wrong
  - Examples: DSA, RSA [PKCS#1]
- Signing is not encryption with public key!
  - Common misconception because the RSA private key can be used both to sign and decrypt
- Digital signature "with appendix"
  - Hash the message, sign the hash
  - The signature is usually appended to the actual message but can also be stored separately
- Question: what consequences if you use a broken hash function with known collisions (e.g. SHA-1) for signing?

PUBLIC-KEY ENCRYPTION

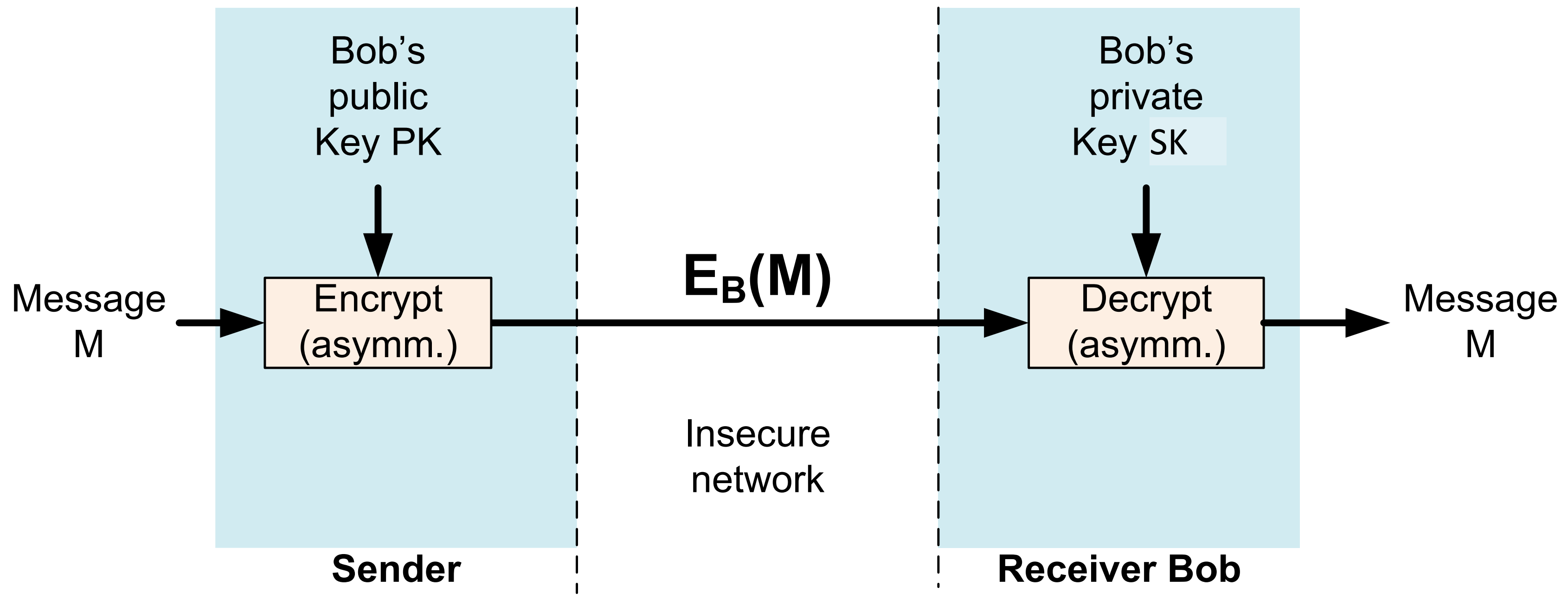
Public-key encryption



- Asymmetric encryption: public key and private key
- Protects secrets, not integrity

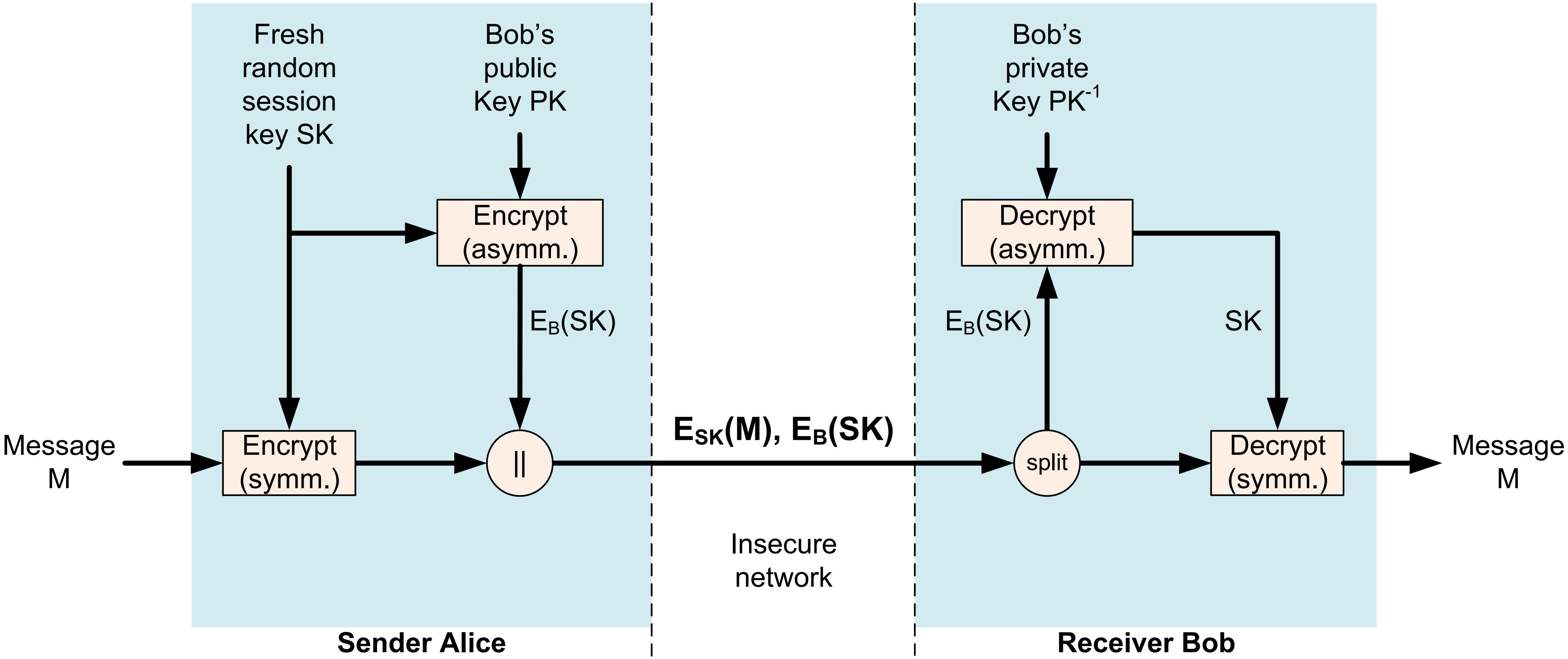
[Daemon graphics: www.freebsd.org]

Public-key encryption



- Message encryption based on asymmetric cryptography
  - Key pair: public key and private key
- Protects secrets, not integrity

Hybrid encryption



- Symmetric encryption is fast; asymmetric is convenient
- Hybrid encryption = symmetric encryption with random session key + asymmetric encryption of the session key

Key distribution

!

- The advantage of public-key cryptography is easier key distribution
- Shared secret keys, symmetric cryptography:
  - $O(N^2)$ pairwise keys for N participants → does not scale
  - Keys must be kept secret → hard to distribute safely
- Public-key protocols, asymmetric cryptography:
  - N key pairs needed, one for each participant (or 2·N if different key pairs for encryption and signature)
  - Public keys are public → can be posted on the Internet

But… both shared and public keys must be authentic

How does Alice know she shares K_{AB} with Bob, not with Eve?

How does Alice know PK_B is Bob's public key, not Eve's?

RSA encryption details

- RSA encryption, published 1978
  - Based on modulo arithmetic with very large integers
- Simplified description of the algorithm:
  - Public key (e,n) - public exponent and modulus
  - Private key (d,n) - secret exponent and public modulus
  - Encryption $C = M^e \bmod n$
  - Decryption $C^d \bmod n = (M^e)^d \bmod n = M$
  - n is commonly 1024 or 2048 bits long, d will also be long,
    e can be short (17 or $2^{16}+1$); M can be at most as long as n

- Why does it work? Based on number theory
  - Euler's totient function $\varphi(n)$, number of integers 1...n that are relatively prime with n
  - Euler's theorem: $x^{\varphi(n)} \equiv 1 \pmod n$, and thus $x^{k\varphi(n)+1} \equiv X \pmod n$
  - We need to have e and d so that $ed = k\varphi(n)+1$ for some k
  - Key pair generation:
    1. Choose n as product of two large secret prime numbers $n=pq$; then, $\varphi(n)=(p-1)(q-1)$
    2. Then pick a small e ($e=17$ or $e=2^{16}+1$), solve d with the extended Euclidian algorithm
    3. Forget $p,q,\varphi(n)$
  - RSA security assumption: difficult to solve d when you only know (e,n)
    (this is assumed to be about as difficult as factoring n without being told p and q)

- For details and implementation guidelines, see PKCS#1
  Never implement RSA without following such a standard!

Example: RSA public key

ASN.1 type tags

```
30 82 01 0a 02 82 01 01 00 c7 3a 73 01 f3 2e a8
72 25 3c 6b a4 14 54 24 e7 e0 ab 47 2e 9f 38 a7
12 77 dc cf 62 bc de 47 a2 55 34 a6 47 9e d6 13
90 3d 9f 72 aa 42 32 45 c4 4a b7 88 cc 7b c5 a6
18 4f d5 86 a4 9e fb 42 5f 37 47 53 e0 ff 10 2e
cd ed 4a 4c a8 45 d9 88 09 cd 2f 5f 7d b6 9b 40
41 4f f7 a9 9b 7a 95 d4 a4 03 60 3e 3f 0b ff 83
d5 a9 3b 67 11 59 d7 8c aa be 61 91 d0 9d 5d 96
4f 75 39 fb e7 59 ca ca a0 63 47 bd b1 7c 32 27
1b 04 35 5a 5e e3 29 1a 06 98 2d 5a 47 d4 05 b3
22 3f fd 43 38 51 20 01 ad 1c 9e 4e ad 39 f4 d1
ae 90 7d f9 e0 81 89 d2 b7 ba cd 68 2e 62 b3 d7
ad 00 4c 52 24 29 97 37 8c 6e 36 31 bd 9d 3d 1d
4c 4c cc b0 b0 94 86 06 9c 13 02 27 c5 7c 1e 2e
f6 e3 f6 13 37 d9 fb 23 9d e7 c7 d5 ce 94 54 7d
ef ef df 7b 7b 79 2e f9 75 37 8a c1 ef a5 c1 2a
01 e0 05 36 26 6a 98 bb d3 02 03 01 00 01
```

2048-bit modulus

ASN.1 type tags

public exponent ($2^{16}+1$)

Key length in asymmetric crypto

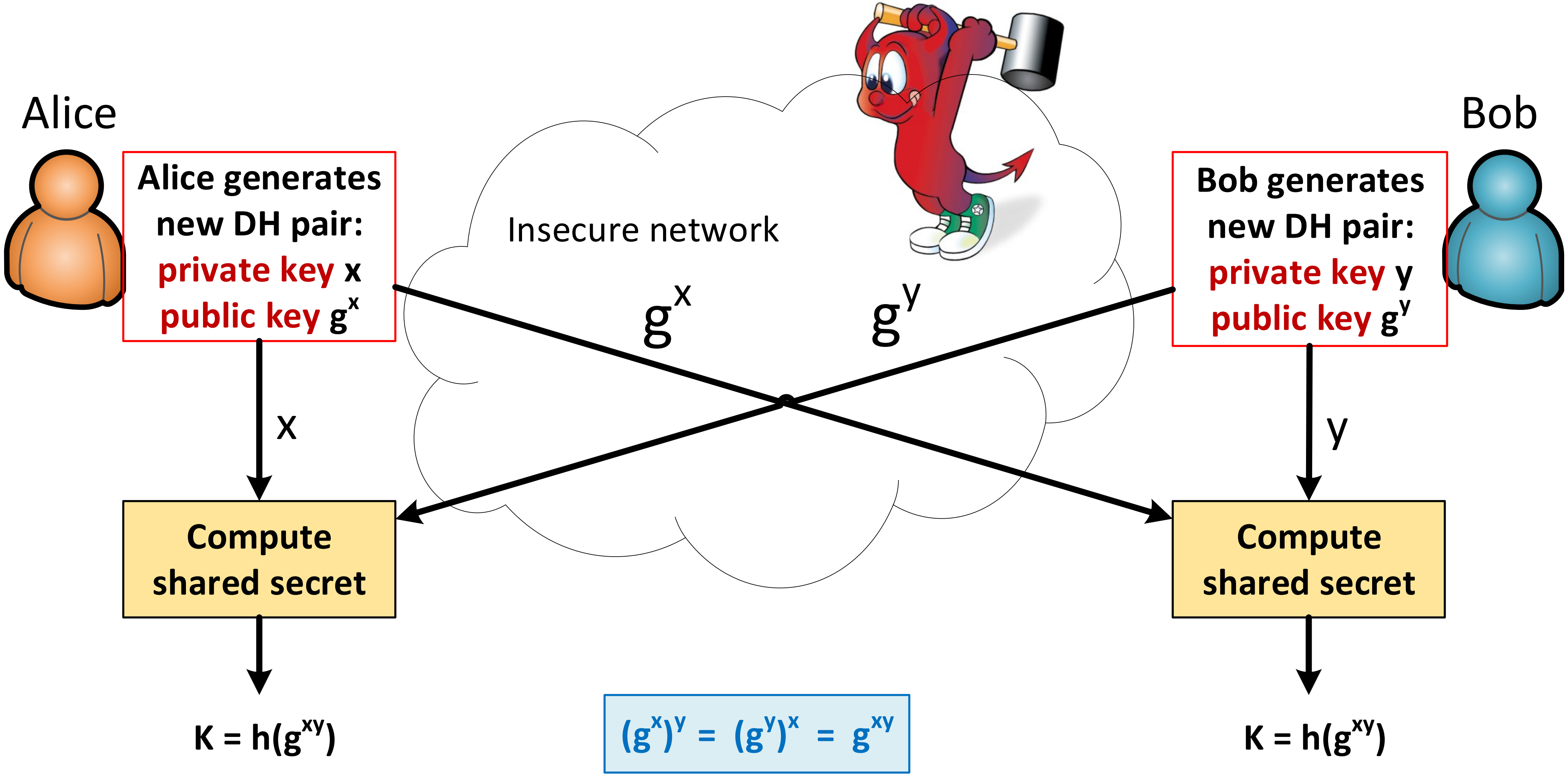
- In RSA, secure key lengths are ≥ 2048 bits
- Elliptic-curve cryptography (ECC):
  public-key crypto with much shorter keys and efficient computation, ≥ 256 bits
  – Used for most new applications and small devices

Formal security definitions

- Cryptographic security definitions for asymmetric encryption
- Semantic security (security against passive attackers)
  - Computational security against a ciphertext-only attack
- Ciphertext indistinguishability (active attackers)
  - IND-CPA — attacker submits two plaintexts, receives one of them encrypted, and is challenged to guess which it is ⇔ semantic security
  - IND-CCA — indistinguishability under *chosen ciphertext* attack i.e. attacker has access to a decryption oracle before the challenge
  - IND-CCA2 — indistinguishability under *adaptive* chosen ciphertext attack i.e. attacker has access to a decryption oracle before and after the challenge (except to decrypt the challenge)
- Non-malleability
  - Attacker cannot modify ciphertext to produce a related plaintext
  - NM-CPA ⇒ IND-CPA; NM-CCA2 ⇔ IND-CCA2
- It is non-trivial to choose the right kind of encryption for your application; ask a cryptographer!

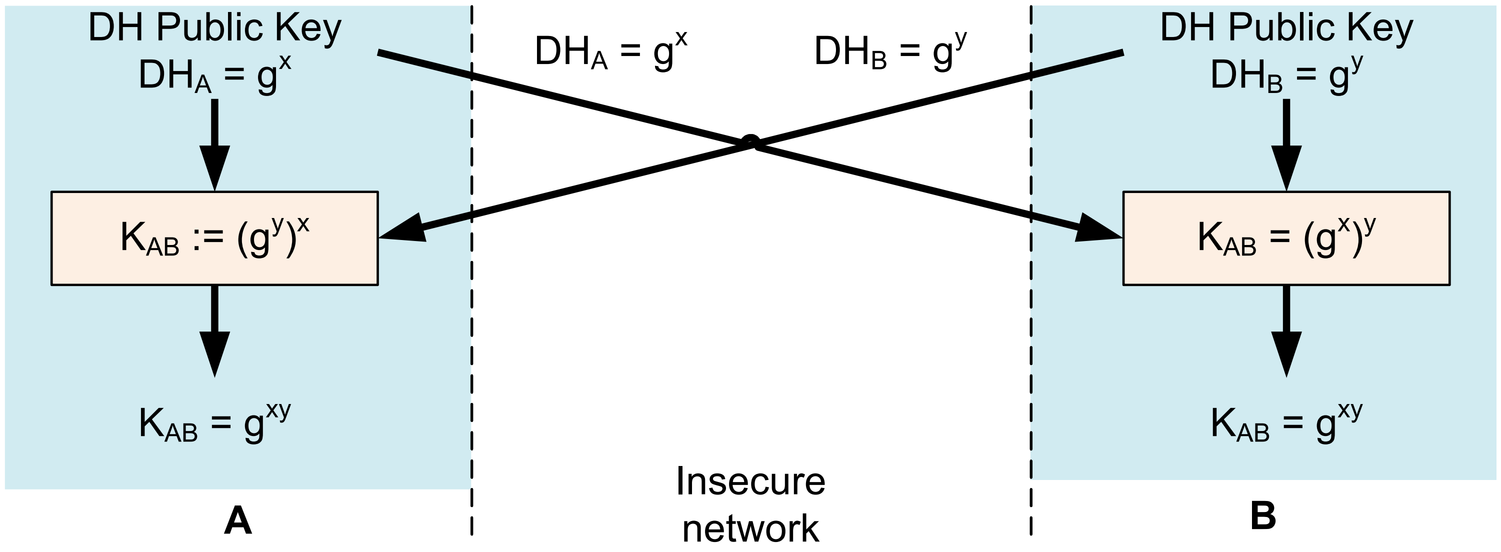
DIFFIE-HELLMAN KEY EXCHANGE

Diffie-Hellman key exchange



[Daemon graphics: www.freebsd.org]

Diffie-Hellman key exchange



- Both sides compute the same session key
- Passive attacker listens to communication but cannot compute the key

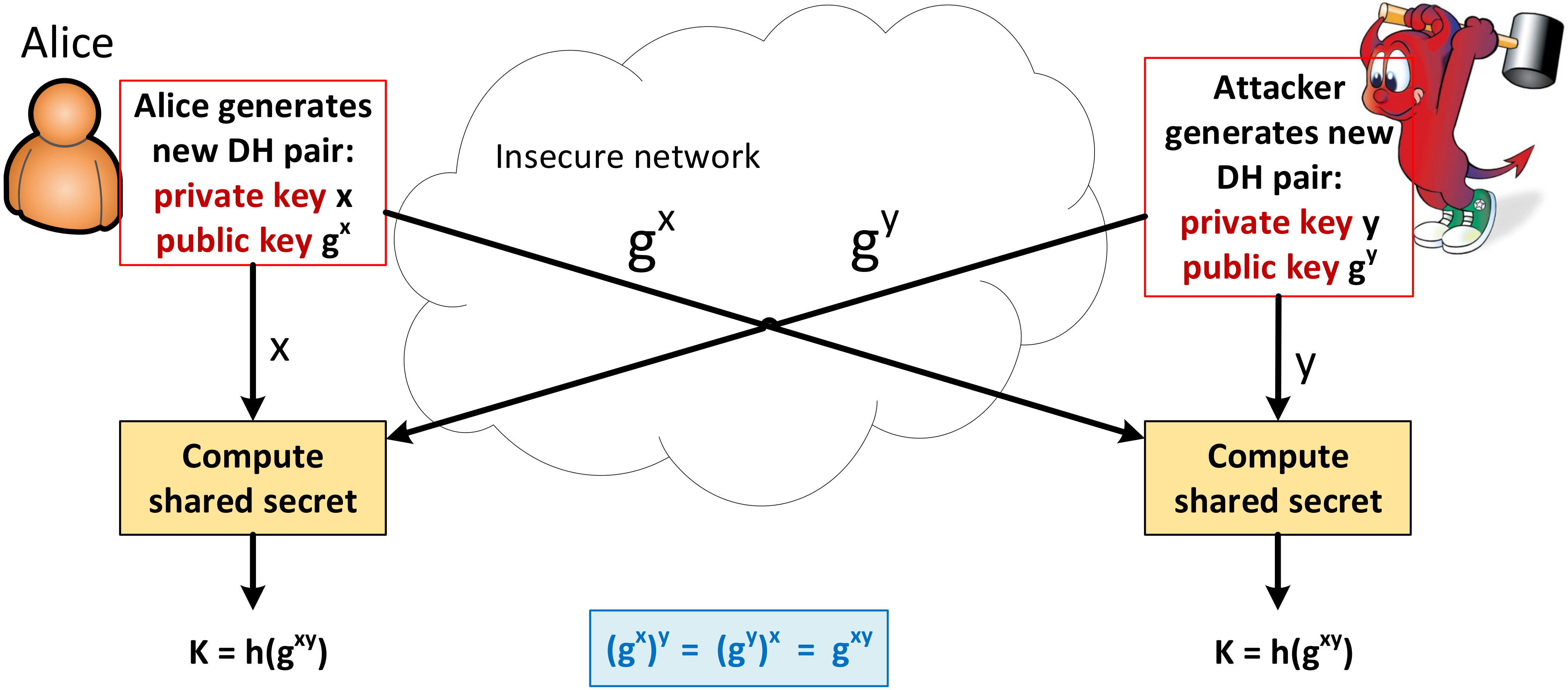
Diffie-Hellman key exchange

- Creating a shared key based on commutative operation, such as exponentiation modulo p:

$$(g^x \bmod p)^y \bmod p \quad = \quad (g^y \bmod p)^x \bmod p$$

- Diffie-Hellman assumption: given g, p, g^x and g^y, it is infeasible to solve g^{xy}
  - Security depends on the difficulty of the discrete logarithm problem, i.e. solving x from (g^x mod p) when p is large
- Elliptic curve Diffie-Hellman uses commutative operations in a different field

Impersonation attack



[Daemon graphics: www.freebsd.org]

Authenticated Diffie-Hellman

- Diffie-Hellman key exchange is vulnerable to impersonation attacks: Shared secret key, ok, but with whom?
  Without authentication, it could be anyone.
- Unauthenticated DH is secure against passive attackers who only listen, but not against active attackers who also lie and pretend
- Solution: authenticate the key-exchange messages
  - Sign with public-key signatures
  - Compare manually between endpoints

SUMMARY

How strong is cryptography?

- Cryptology viewpoint: requires continuous analysis and improvement
- Engineering viewpoint: unbreakable for years if you use strong standard algorithms and 128..256-bit symmetric keys
  - May need to upgrade algorithms every 10 years or so
  - Avoid using algorithms in creative ways that are not their original purpose
- Weak crypto is worse than no crypto, use strong algorithms and keys
- Which algorithms can be trusted?
  - Block ciphers have endured relatively well, hash functions require upgrading
  - Quantum computers might break public-key cryptography
- Almost no absolute proofs of security exist!

Security vs. cryptography

- Cryptography: mathematical methods for encryption and authentication
- In this course, we use cryptography as one building block for security mechanisms
- Remember that cryptography alone does not solve all security problems:

  "Whoever thinks his problem can be solved using cryptography, doesn't understand the problem and doesn't understand cryptography."
  — attributed to Roger Needham and Butler Lampson

Message size overhead

- Authentication increases the message size:
  - MAC or signature is appended to the message
  - MAC takes 16–32 bytes
  - 4096-bit RSA signature is 512 bytes
  - Elliptic-curve signatures (ECDSA) can be 64..128 bytes
- Encryption increases the message size:
  - In block ciphers, messages are padded to nearest full block
  - IV for block cipher takes 8–16 bytes
  - 1024-bit RSA encryption of the session key is 128 bytes
- Overhead of headers, type tags etc.
- Small size increase ok for most applications but can cause problems in some:
  - Signing individual IP packets (1500-byte limit on packet size)
  - Authenticating small wireless frames
  - Encrypting file system sector by sector, but cannot increase sector size by a few bytes to fit in the IV or MAC

List of key concepts

- Cryptographic hash function, pseudorandom, preimage resistance, second-preimage resistance, collision resistance, birthday attack, MAC, HMAC
- Symmetric cryptography, shared secret key, key length, encryption, decrypting, plaintext, ciphertext, Kerckhoff's principle, block cipher, cipher mode, AES, CBC mode, authenticated encryption, AES-GCM
- Asymmetric or public-key cryptography, kay pair, public key, private key, RSA, elliptic-curve cryptography ECC, hybrid encryption, digital signature, key distribution, Diffie-Hellman key exchange, ECDH
- Message secrecy or confidentiality, integrity, authentication, weak and strong cryptography, impersonation

Notations in protocol specifications and research papers

- Shared key:
  $K = SK = K_{AB}$
- Symmetric encryption:
  $Enc_K(M)$, $E_K(M)$, $E(K;M)$, $\{M\}_K$, $K\{M\}$
- Hash function:
  $h(M)$, $H(M)$, $hash(M)$, $SHA\text{-}256(M)$
- Message authentication code:
  $MAC_K(M)$, $MAC(K;M)$, $HMAC_K(M)$
- Public/private key:
  $PK = PK_A = K_A = K^+ = K^+_A = e$; $SK = PK^{-1} = PK^{-1}_A = K^- = K^-_A = d$
- Public-key encryption:
  $Enc_B(M)$, $E_B(M)$, $PK\{M\}$, $\{M\}_{PK}$
- Signature notations:
  $S_A(M) = Sign_A(M) = S(PK^{-1}; M) = PK^-_A(M) = \{M\}_{PK-1}$