CT437
COMPUTER SECURITY AND FORENSIC COMPUTING

PUBLIC KEY CRYPTOGRAPHY

Dr. Michael Schukat

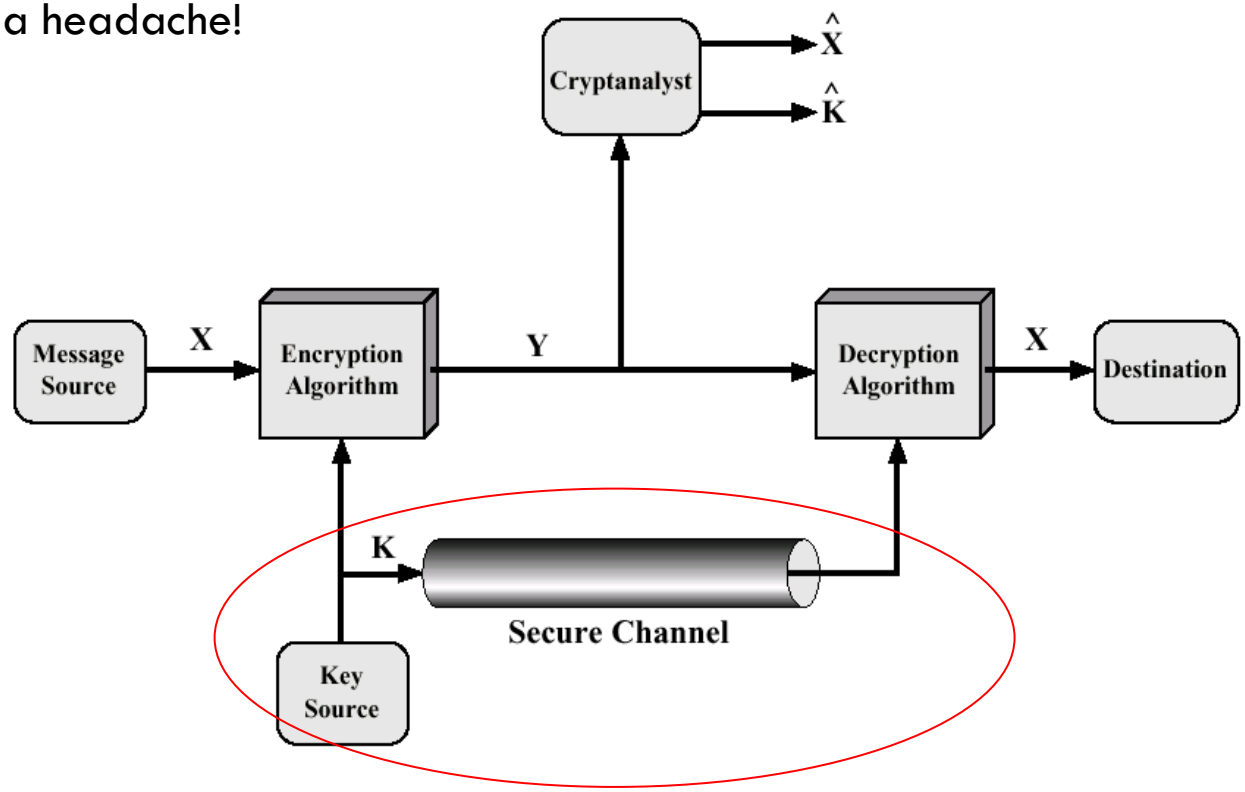
Ollscoil na Gaillimhe
University of Galway

Lecture Content

- Public key cryptography versus private key cryptography
- Public key cryptography applications
- Diffie-Hellman Key exchange
  - Man-in-the-Middle (MitM) attacks
- RSA encryption
- Optimisation techniques for public key encryption
- ECC encryption
- The Double-Ratchet algorithm

Model of Conventional Cryptosystem

Symmetric block ciphers are cryptographically strong, but key distribution can be a headache!



$$Y = E_K(X), X = E_K^{-1}(Y)$$

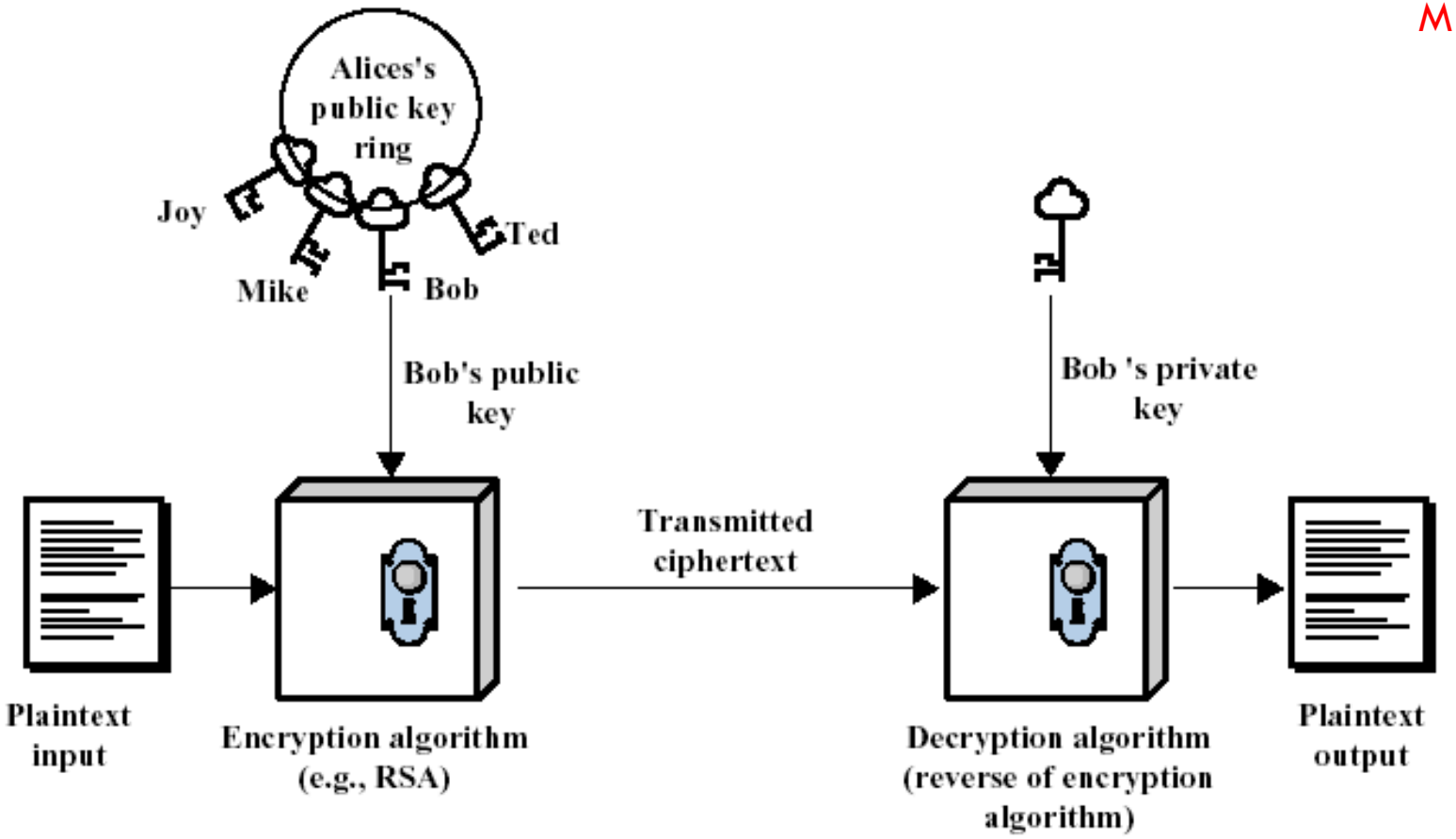
Features and Limitations of Private-Key Cryptography

- Traditional symmetric/single key cryptography uses one key, shared by both sender and receiver
  - If this key is disclosed, communications are compromised
- The key is also symmetric, both parties are equal
  - This is problematic too, as it does not protect the sender from a situation, where:
    - the receiver forges a message using that key
    - and claims that it was sent be the sender
    - Think about an electronic contract that is exchanged between two business partners that use a shared key
    - One party can forge a contract and claim it was sent by the other side
    - Message authentication (HMAC or CMAC) doesn't solve the problem!

Features of Public-Key Cryptography

- **Public-key/two-key/asymmetric cryptography** involves the use of two keys:
  - a **public-key**, which the owner shares with any peer; it is used to:
    - Encrypt messages send from the peer to the owner
    - Verify the integrity and origin of messages send from the owner to a peer (signature validation)
  - a **private-key**, known only to the recipient/owner, used to:
    - Decrypt messages that were encoded using their public key
    - Digitally sign data send to a peer (signature creation)
- The keys are **asymmetric**, because they are not equal
- Those who encrypt a message or verify a signature (using the receiver's public key) cannot decrypt the message or forge a signature
- It is computationally very hard (and infeasible) for an attacker to rebuild an owner's private key by analysing their public key
- This is achieved through the application of number- theoretic concepts

Public-Key Encryption

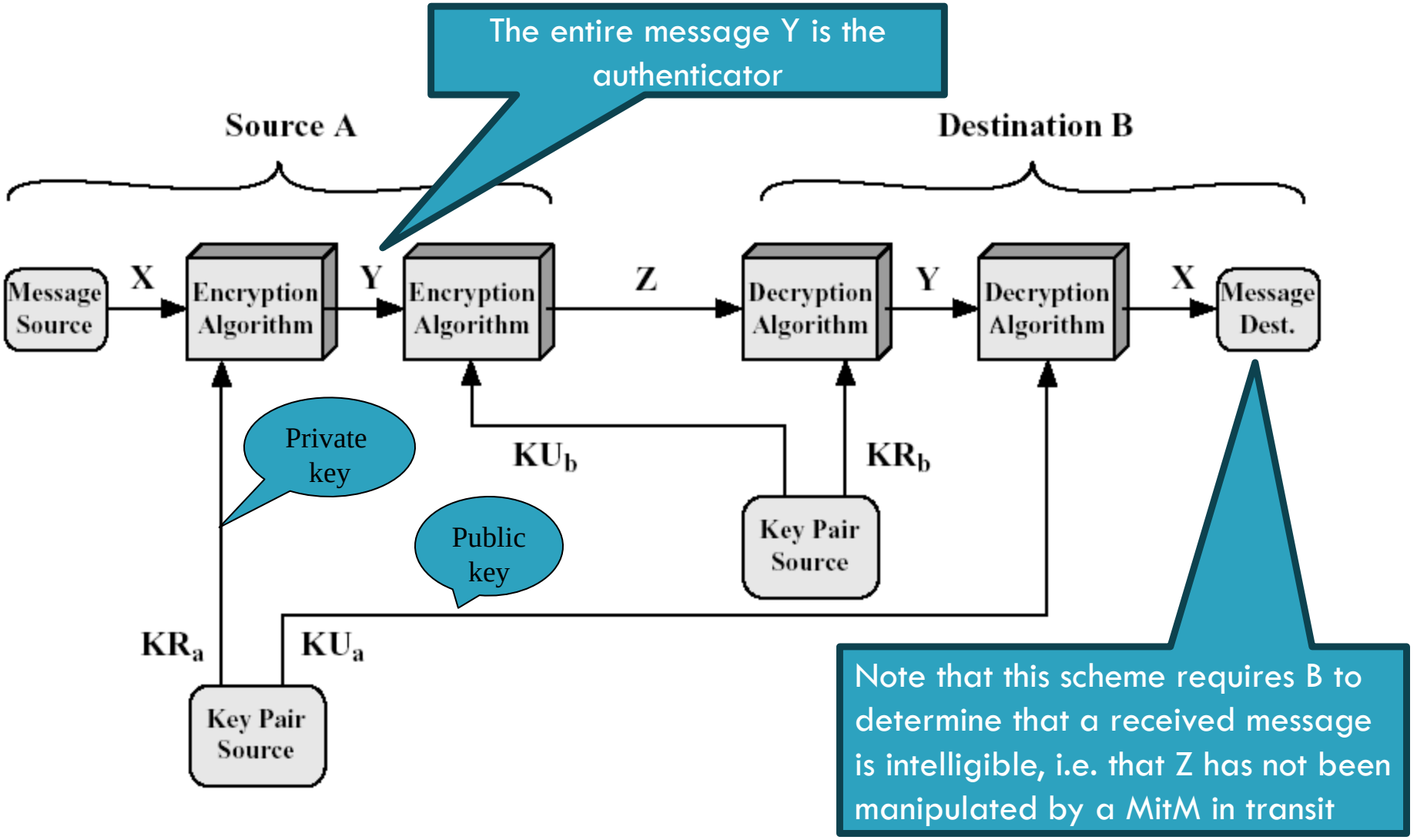


Applications of Public-Key Cryptosystems

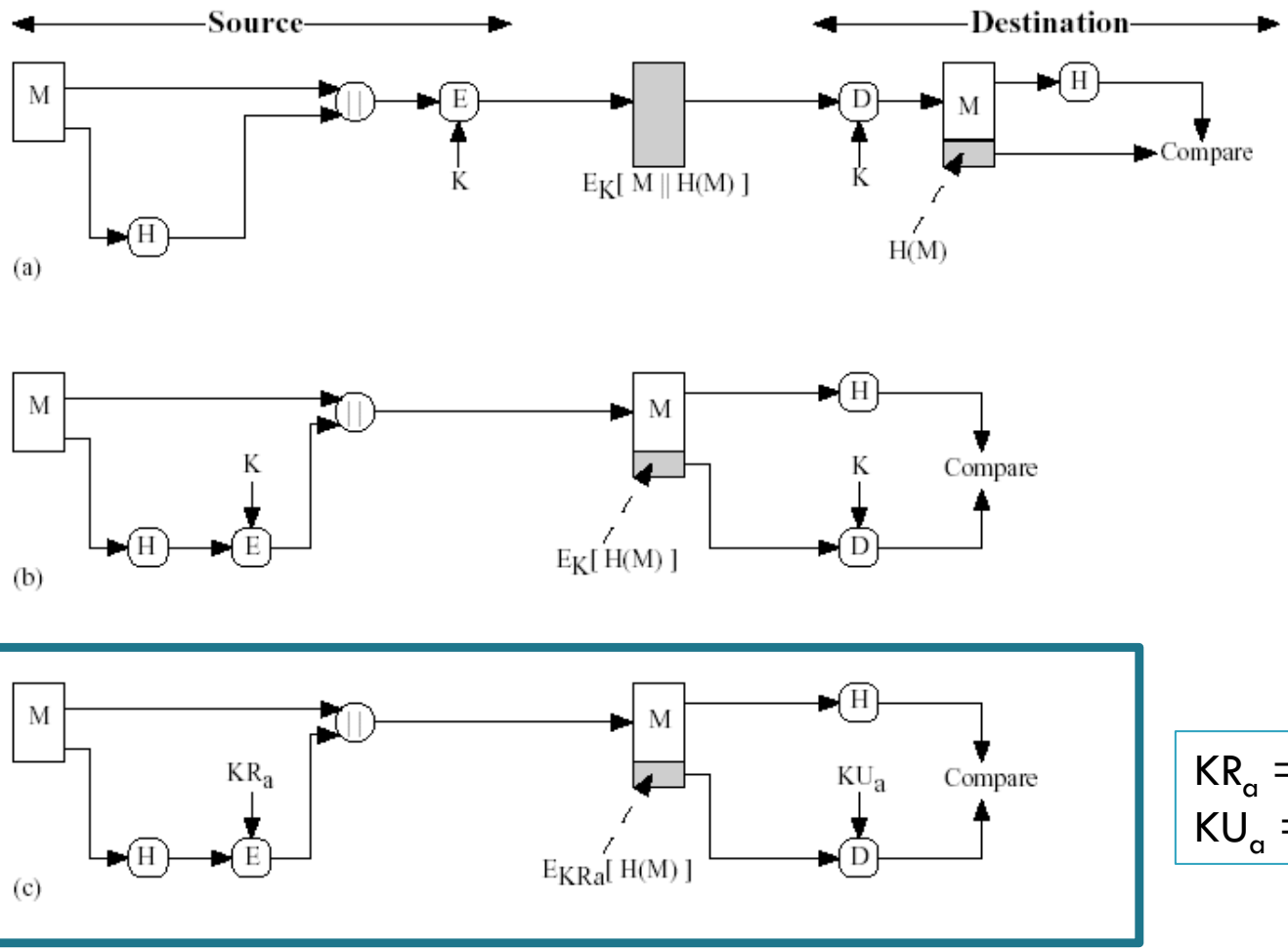
- **Data encryption/decryption:**
  The sender encrypts the message with the recipient's public key and the receiver decodes the message using their private key
  - Recall symmetric encryption where only **one** key is used
- **Digital signature/authentication:**
  The sender "signs" a message with their private key. Signing is achieved by encrypting the message or its MAC using their private key (next slide)
  - Recall private key encryption where sender and receiver just share one key
- **Key exchange:**
  Two sides negotiate a **symmetric** session key
  - Private key encryption is much faster than public key encryption
  - This key may also be used for conventional message authentication

- Note that in order to avoid confusion we use from now on the terms:
  - Symmetric key for private key encryption (block ciphers and stream ciphers)
  - Public and private keys for public key encryption

Public-Key Cryptosystems: Secrecy and Authentication

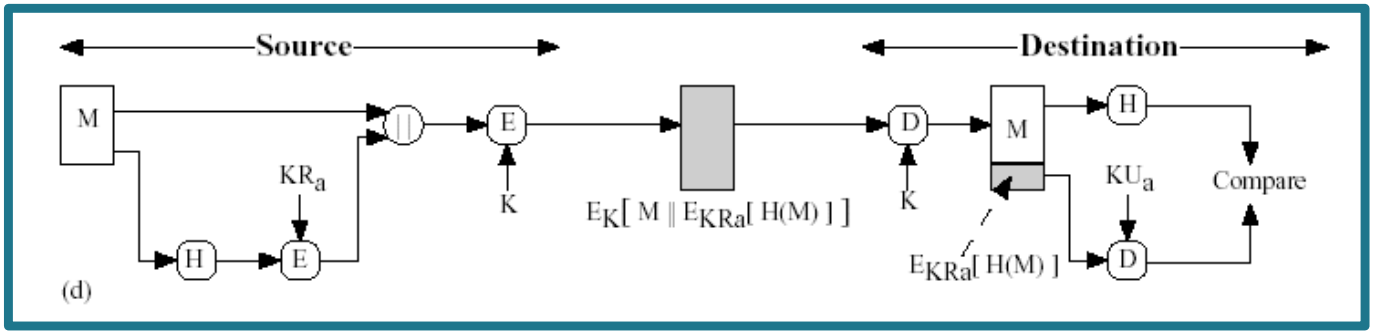


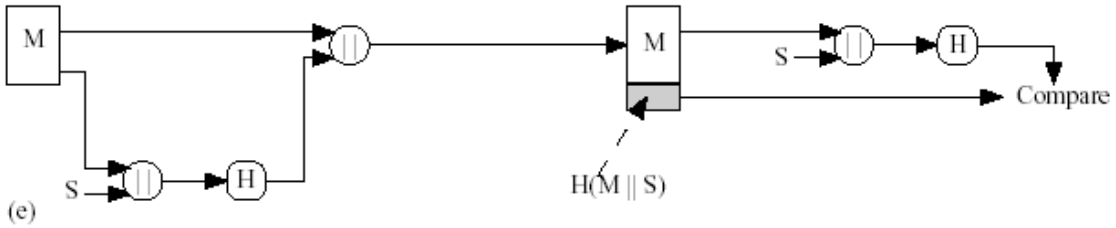
Recap: Basic Uses of Hash Functions (H) in Combination with asymmetric Encryption (c)

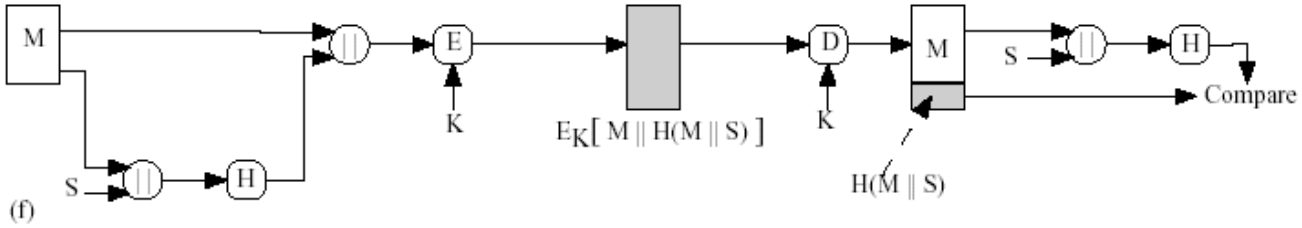


KR_a = Sender's private key
KU_a = Sender's public key

Recap: Basic Uses of Hash Functions (H) in Combination with asymmetric Encryption (d)







KR_a = Sender's private key
KU_a = Sender's public key

Public-Key Cryptosystems

- There are different cryptosystems, including (from simplest to most complex):
  - **Diffie Hellman key exchange**
  - **RSA**
  - DSS
  - **Elliptic Curve Cryptography**

Algorithm	Encryption/Decryption	Digital Signature	Key Exchange
RSA	Yes	Yes	Yes
Diffie-Hellman	No	No	Yes
DSS	No	Yes	No

Modular Arithmetic

- Modular arithmetic is a system of arithmetic for integers, where numbers wrap around when reaching a certain value n, called the modulus
  - Recall modulus operator "%" in C and other languages, i.e. "division with rest" with rest being the modulus
  - Example: 75 / 6 = 12 remainder 3 ➔ 75 % 6 = 3
- Numbers {0, 1, …, n - 1} are called "multiplicative group of integers modulo n", or simply Z_n, for some n > 0
- Within Z_n, addition and multiplication is well defined!

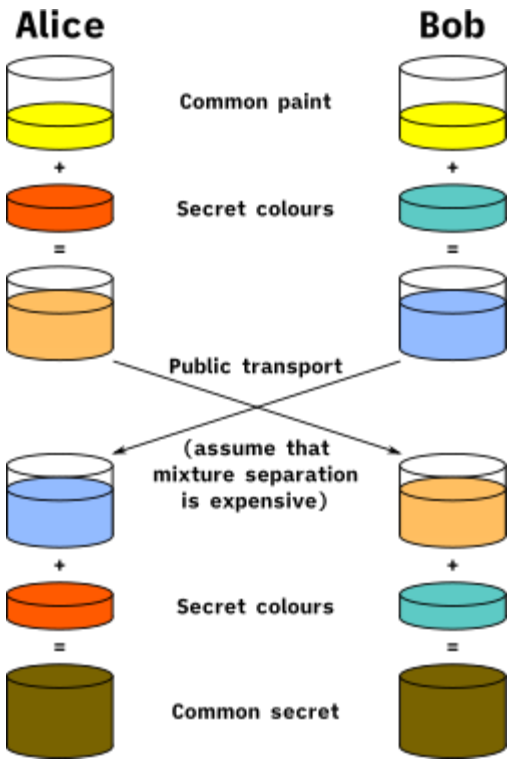
Example: Multiplication in Z_9

Mx3

*	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8
2	0	2	4	6	8	1	3	5	7
3	0	3	6	0	3	6	0	3	6
4	0	4	8	3	7	2	6	1	5
5	0	5	1	6	2	7	3	8	4
6	0	6	3	0	6	3	0	6	3
7	0	7	5	3	1	8	6	4	2
8	0	8	7	6	5	4	3	2	1

Illustration of Concept behind Diffie-Hellman Key Exchange (Wikipedia)

- Alice and Bob want to share a secret colour using public transport
  - i.e. an adversary (i.e. Mallory, not shown) can get samples of any colour that is exchanged between both
- Alice and Bob agree on a common "public" paint color (yellow in the example)
- Each of them add a secret colour and send their mix to the other party
  - Mallory can intercept both, but cannot separate the mixtures
- Alice and Bob receive the other's mixture and add their secret colour
- Both colours are identical
- → This color is their common secret



Diffie-Hellman Key Exchange

- Diffie-Hellman provides a mechanism for a secure key exchange between two endpoints
  - The negotiated key is subsequently used as a symmetric key (or as a seed for a key) for data encryption and message authentication (as seen before)
- The algorithm uses the multiplicative group of integers modulo q
  - q has typically a length of 1024 or 2048 bits
- It is based on the difficulty of computing discrete logarithms over such groups, e.g.

$$6^3 \bmod 17 = 216 \bmod 17 = 12 \qquad \text{(easy)}$$

$$12 = 6^y \bmod 17? \qquad \text{(difficult)}$$

  - Recall $6^3 = 6 \times 6 \times 6$, so we need just the multiplication

- The core equation for the key exchange is

$$K = (A)^B \bmod q$$

Diffie-Hellman: Global Public Elements

- Alice and Bob select:
  - A prime number q which determines Z_q
  - A positive integer a, with $1 < a < q$ and a is a **primitive root** of q
    - Note that a is also called the generator
- Definition: a is a primitive root of q, if numbers
  $a \bmod q,\ a^2 \bmod q,\ \cdots\ a^{(q-1)} \bmod q$
  are distinct integer values between 1 and $(q - 1)$ (i.e. in Z_q) in some permutation

- Example: $a = 3$ is a primitive root of Z_5 (i.e. q = 5), $a = 4$ is not:

$3^1 = 3 = 0 * 5 + \mathbf{3}$
$3^2 = 9 = 1 * 5 + \mathbf{4}$
$3^3 = 27 = 5 * 5 + \mathbf{2}$
$3^4 = 81 = 16 * 5 + \mathbf{1}$

$4^1 = 4 = 0 * 5 + \mathbf{4}$
$4^2 = 16 = 3 * 5 + \mathbf{1}$
$4^3 = 64 = 12 * 5 + \mathbf{4}$
$4^4 = 256 = 51 * 5 + \mathbf{1}$

Primitive Roots of Z_n with $15 < n < 32$

n	primitive roots modulo n
16	
17	3, 5, 6, 7, 10, 11, 12, 14
18	5, 11
19	2, 3, 10, 13, 14, 15
20	
21	
22	7, 13, 17, 19
23	5, 7, 10, 11, 14, 15, 17, 19, 20, 21
24	
25	2, 3, 8, 12, 13, 17, 22, 23
26	7, 11, 15, 19
27	2, 5, 11, 14, 20, 23
28	
29	2, 3, 8, 10, 11, 14, 15, 18, 19, 21, 26, 27
30	
31	3, 11, 12, 13, 17, 21, 22, 24

Generation of Secret-Key: Part 1

- Alice and Bob share publicly a prime number q and a primitive root a
- Alice (User A):
  - Select secret number XA with $0 < XA < q$
  - Calculate public value $YA = a^{XA} \bmod q$ (← difficult to reverse)
  - YA is sent to Bob (user B)
- Bob (User B):
  - Select secret number XB with $0 < XB < q$
  - Calculate public value $YB = a^{XB} \bmod q$ (← difficult to reverse)
  - YB is send to Alice

Generation of Secret-Key: Part 2

- Alice:
  - Alice owns X_A and receives Y_B
  - She generates the secret key: $K = (Y_B)^{X_A} \bmod q$
- Bob:
  - Bob owns X_B and receives Y_A
  - Bob generates the secret key: $K = (Y_A)^{X_B} \bmod q$
- **Both keys are identical!**

Generation of Secret-Key: Part 2

$$K = (YB)^{XA} \bmod q$$

$$= (a^{XB} \bmod q)^{XA} \bmod q$$

$$= (a^{XB})^{XA} \bmod q$$

$$= a^{XB\ XA} \bmod q$$

$$= a^{XA\ XB} \bmod q$$

$$= (a^{XA})^{XB} \bmod q$$

$$= (a^{XA} \bmod q)^{XB} \bmod q$$

$$= (YA)^{XB} \bmod q$$

Example for Diffie-Hellman

- Alice and Bob agree on public values q and a, and determine their respective secrets XA and XB :
- Let $q = 5$ and $a = 3$
- Alice picks $XA = 2$, therefore $YA = a^{XA} \bmod 5 = 4$
- Bob picks $XB = 3$, therefore $YB = a^{XB} \bmod 5 = 2$
- Alice sends $YA = 4$ to Bob
- Bob sends $YB = 2$ to Alice
- Alice calculates: $K = (YB)^{XA} \bmod q = 2^2 \bmod 5 = 4$
- Bob calculates: $K = (YA)^{XB} \bmod q = 4^3 \bmod 5 = 4$

Ephemeral versus Static Diffie-Hellman Keys

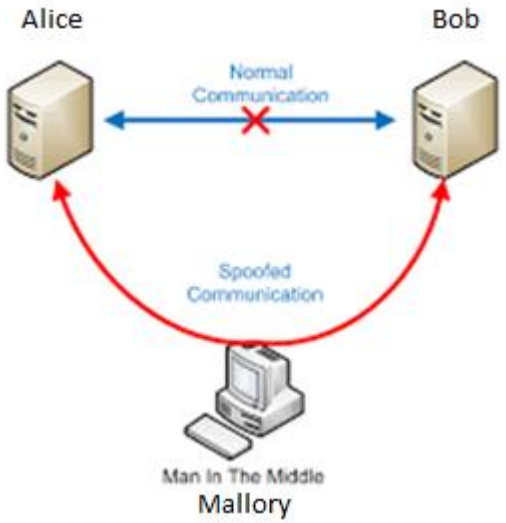
- The generated DH keys can be either
  - static (to be reused)
  - ephemeral (only used once, e.g., for one session only)
- Ephemeral keys
  - provide forward secrecy, but no endpoint authenticity
    - Forward secrecy: If the current key is recovered by an adversary, it only effects the current session, but no past or future sessions
- Static keys
  - do not provide forward secrecy
  - do provide (implicit) endpoint authenticity
  - do not protect against replay-attacks

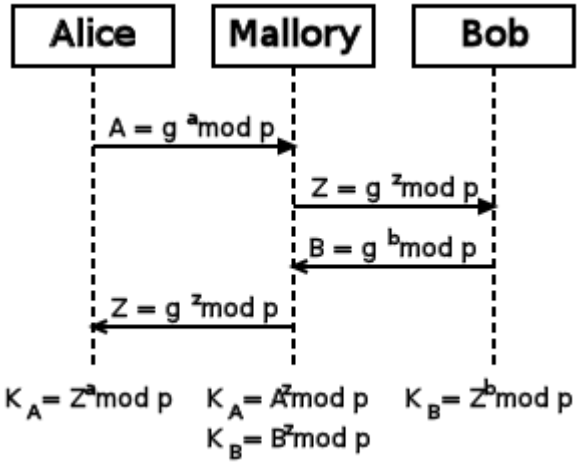
Example DH Parameters

- Standardised, see https://www.ietf.org/rfc/rfc3526.txt
- Example 2048-bit MODP Group
  - q = 2^2048 - 2^1984 - 1 + 2^64 * { [2^1918 pi] + 124476 }
  - q = FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1 29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245 E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE45B3D C2007CB8 A163BF05 98DA4836 1C55D39A 69163FA8 FD24CF5F 83655D23 DCA3AD96 1C62F356 208552BB 9ED52907 7096966D 670C354E 4ABC9804 F1746C08 CA18217C 32905E46 2E36CE3B E39E772C 180E8603 9B2783A2 EC07A28F B5C55DF0 6F4C52C9 DE2BCBF6 95581718 3995497C EA956AE5 15D22618 98FA0510 15728E5A 8AACAA68 FFFFFFFF FFFFFFFF
  - a = 2

[] == rounded

DH and Man-in-the-Middle (MitM) Attacks





- Mallory is a MitM attacker with the ability to intercept, and fabricate messages
  - Not to confuse with a Meet-in-the-Middle attack (→ double-DES and triple-DES)
- Both Alice and Bob are unaware of Mallory's existence, as there is no mutual authentication and an unprotected communication link
- Alice and Bob exchange their shared values (A and B in the example), but these are intercepted by Mallory
- Mallory completes both key exchanges sending her own shared value Z to both Alice and Bob
- By doing so, Mallory establishes two individual (secure) connections with Alice and Bob
- Alice and Bob have no idea that they became victims of a MitM attack!

In-Class Activity: Diffie-Hellman MitM Attack

- Let q = 5 and a = 3;
- $X_{Alice} = 2$, therefore $Y_{Alice} = a^{XAlice} \bmod 5 = 4$
- $X_{Bob} = 3$, therefore $Y_{Bob} = a^{XBob} \bmod 5 = 2$
- $X_{Malory} = 1$, therefore $Y_{Malory} = a^{XMalory} \bmod 5 = 3$
- What session keys between
  - Alice and Malory
  - Malory and Bob

  are generated?
- Note: User A's key $K = (YB)^{XA} \bmod q$
- Note: User B's key $K = (YA)^{XB} \bmod q$

Solution

- Alice sends "4" to Bob, but this message is intercepted by Malory
- Bob sends "2" to Alice, but this message is intercepted by Malory
- Malory sends "3" to both parties, claiming to be either Bob or Alice
- Alice receives "3" and calculates K as follow: $K = 3^2 \bmod 5 = 4$
  - Malory calculates $4^1 \bmod 5 = 4$
- Bob receives "3" and calculates K as follow: $K = 3^3 \bmod 5 = 2$
  - Malory calculates $2^1 \bmod 5 = 2$
- Alice and Bob think they just mutually agreed on a shared secret key
- From this point onwards Malory as a MitM can read, manipulate and fabricate messages between Alice and Bob

The RSA Algorithm

- Published by Rivest, Shamir and Adleman in 1977, but first discovered by Clifford Cocks (British mathematician and cryptographer) in 1973
- The RSA scheme works similar to a block cipher, where a plaintext M and a ciphertext C are integers between 0 and $n - 1$, i.e. elements of Z_n
- M can be a plaintext message (block), a hash value, or a private key picked by the sender to be shared with the message recipient
  - E.g., "ABC" = "01000001 01000010 01000011" = 4276803_{10}
- Principle: $C = M^e \bmod n$
  $M = C^d \bmod n = M^{ed} \bmod n$
- Public key $KU = \{e, n\}$
- Private key $KR = \{d, n\}$
- With n sufficiently large it is infeasible to determine d given e and n

Key Generation for the RSA Algorithm

Euler's totient function Phi

Key Generation

Select p, q	p and q both prime
Calculate $n = p \times q$	
Calculate $\phi(n) = (p - 1)(q - 1)$	
Select integer e	$\gcd(\phi(n), e) = 1; \quad 1 < e < \phi(n)$
Calculate d	$d = e^{-1} \bmod \phi(n)$
Public key	$KU = \{e, n\}$
Private key	$KR = \{d, n\}$

Greatest common divisor

See next slide

Example

- Let $p = 7,\ q = 11$ and $n = pq = 77$
- $\phi(77) = (p - 1)(q - 1) = 6 \times 10 = 60$
- Factorisation of $60 = 1 * 2 * 5 * 2 * 3$

  Therefore, the divisors of 60 are: $2,\ 3,\ 5$
- List of all integers x, $1<x<60$, with $GCD(60, x) = 1$:
  $7,\ 11,\ 13,\ 17,\ 19,\ 23,\ 29,\ 31,\ 37,\ 47,\ 49,\ 53,\ 59$
- Note that these integers either
  - are prime numbers (that cannot share a common divisor with 60), or
  - do not share a common divisor with 60 (i.e., 7 and 49)

Example (continued)

- Let $e = 7$
- Choose d with $ed = 1 \bmod \phi(pq) \Leftrightarrow$
  $7d = 1 \bmod 60 \Leftrightarrow 7d \bmod 60 = 1$

$7*1 \bmod 60 = 7$ $7*2 \bmod 60 = 14$ $7*3 \bmod 60 = 21$

$7*4 \bmod 60 = 28$ $7*5 \bmod 60 = 35$ $7*6 \bmod 60 = 42$

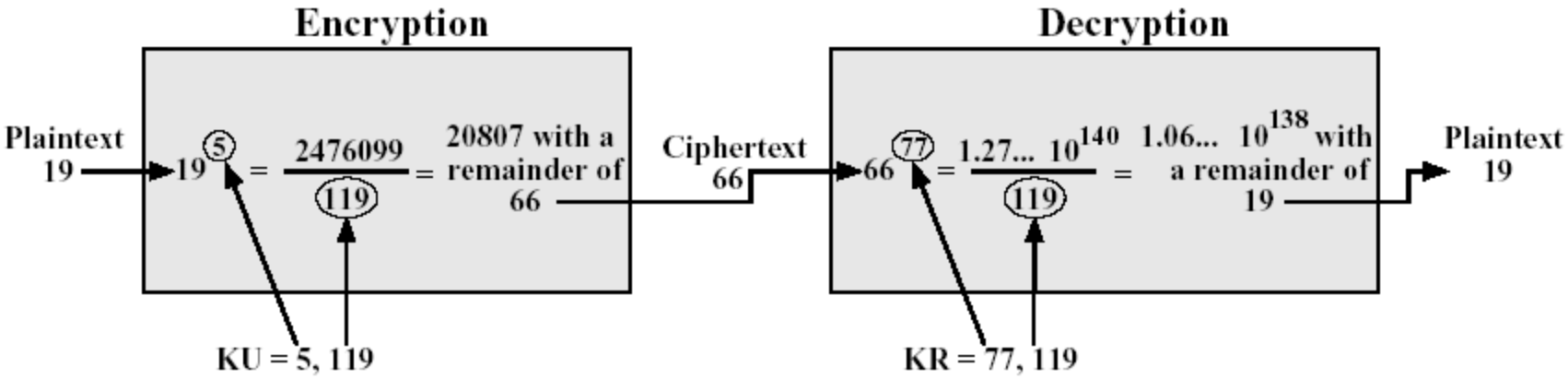
$7*7 \bmod 60 = 49$ $7*8 \bmod 60 = 56$ $7*9 \bmod 60 = 3$

$7*10 \bmod 10 = 28$ $7*11 \bmod 60 = 17$ $7*12 \bmod 60 = 24$

… $7*43 \bmod 60 = 1$

- Therefore $d = 43$
- Therefore $KU = (7, 77)$ and $KR = (43, 77)$
- Note there are better / more efficient algorithms (i.e. the Extended Euclidean Algorithm) to calculate d

Example for an Encryption/Decryption



- Obvious drawbacks:
  - Very large numbers are to be computed
    - Ordinary integer or floating-point variables don't work
    - Instead, large number libraries need to be used
  - This makes RSA encryption / decryption is very slow!

Computational Aspects of Public Key Cryptography

- Assume you have to evaluate the expression $C = 503^{23} \bmod 899$ as part of the encoding process
  - Note that the modulus is small enough to fit into an integer variable
- $503^{23} = 1.367929313795408423250439710106 \times 10^{62}$ cannot be properly represented using an ordinary integer or floating-point variable!
- In order to solve this problem, the exponentiation must be broken down into smaller steps, e.g.
  - $503^{23} \bmod 899 = ((503^{6} \bmod 899) \times (503^{6} \bmod 899) \times (503^{6} \bmod 899) \times (503^{5} \bmod 899)) \bmod 899$
  - $503^{6} \bmod 899 = ((503^{3} \bmod 899) \times (503^{3} \bmod 899)) \bmod 899$
  - $503^{5} \bmod 899 = ((503^{3} \bmod 899) \times (503^{2} \bmod 899)) \bmod 899$
  - $503^{3} \bmod 899 = ((503^{2} \bmod 899) \times 503) \bmod 899$

Computational Aspects of Public Key Cryptography

- … or even iteratively:
  $503^{23} \bmod 899 =$
  $(((((503^2 \bmod 899) \times 503) \bmod 899) \times 503) \bmod 899) \times \cdots \times 503) \bmod 899$

- This expression consists of 22 nested multiplications and 22 nested modulus operations and can be easily calculated by using a loop
- However, once a single number squared is too large to fit into a 32-bit or 64-bit (unsigned) integer variable, a big number library must be used

The Security of RSA

- There are various angles to attack the RSA algorithm:
  - Brute force: Trying all possible private keys (not a great idea!)
  - Mathematical attacks: Factor n (which is the product of two primes); see some very old data below:

Number of Decimal Digits	Approximate Number of Bits	Data Achieved	MIPS-years	Algorithm
100	332	April 1991	7	quadratic sieve
110	365	April 1992	75	quadratic sieve
120	398	June 1993	830	quadratic sieve
129	428	April 1994	5000	quadratic sieve
130	431	April 1996	500	generalized number field sieve

  - See also (for some more recent data) https://en.wikipedia.org/wiki/RSA_numbers#RSA-704
  - Timing attacks: Based on analysis of the run time of an decryption algorithm

Breaking RSA

- Consider the key pair (e, n) and (d, n) or simply (e, n) and d
  - n = p * q, with p and q being large (secret!) primes
- Factorising n is unfeasible for very large n
- However, let's assume n can be factored into p and q
- The adversary can now do the following calculations:
  - $\phi(n) = (p - 1) * (q - 1)$
  - Identify d, so that $e * d = 1 \bmod \phi(n)$
    - e is known, use the aforementioned Extended Euclidean Algorithm

Step 1: Factorise N


```
// This is a very lightweight integer factoring algorithm, not very efficient or
// sophisticated.
// Assume n is the product of two primes p1 and p2
void factorise(int n) {
        int i;
        for (p1 = 2; i <= sqrt(n); i++) {
                if (n % p1 == 0)
                        printf("n = %d; p1 = %d; p2 = %d\n"), n, p1, n / p1);
                break;
}
// Note that the integer values above would be replaced with large number
// representations, i.e., BIGNUM in OpenSSL
```

Step 2: Determine e


```
// We know p and q (n was successfully factorised), d is in the public key KR= d, n
// This is again a very lightweight algorithm, not very efficient or sophisticated.
int breakRSA(int p, int q, int d) {
        int prod, found = 0, start = 1, df = -1;
        int phi = (p -1) * (q – 1);
        while ((!found) && (start < phi)) { // exit if needed
                prod = d * start;
                if (prod % phi == 1) found = 1;
                else start++;
        }
        if (found) df = start;
        return (df);
}
// Note that the integer values above would be replaced with large number
// representations, i.e., BIGNUM in OpenSSL
```

How to choose p and q

- When choosing p and q, the following should be considered:
  1. p <> q, as p = q = sqrt(n)
  2. Neither p or q must not be "small", as factorising could produce a result in a reasonable amount of time (see previous slide "Step 1: Factorise N")
  3. p must not be similar in size to q, because of *Fermat's method of factoring a composite number N*:
     - N can be represented as the difference of two squares:
       - $p * q = \underline{N \Leftrightarrow a^2 - b^2} \Leftrightarrow (a - b)(a + b)$ [== p * q]
     - $N = a^2 - b^2$ can be rewritten as: $b^2 = a^2 - N$
     - To find a solution, iterate through a (starting with round(sqrt(N))), until $a^2 - N$ is a square number (i.e. b^2)

Fermat's Factoring Algorithm


```
// This function assumes N can be factorised. It returns N's factors
// p and q, using "pass by reference" pointers, so that both values
// are returned.
void fermatFactor(int N, int *p, int *q) {
    int a = ceiling(sqrt(N)); // start value for a
    int b2 = a * a - N; // see last slide
    while (sqrt(b2) * sqrt(b2) <> b2) { // is b2 a square?
        a = a + 1; // No, so increment a ...
        b2 = a * a – N; // ... and update b2
    }
    *p = a - sqrt(b2);
    *q = a + sqrt(b2);
}
```

If p (= a - b) and q (= a + b) are similar in size, it takes only a small number of iterations over a to find a solution

Example

1. n = 33 (based on secret values p = 3 and q = 11)
2. First iteration: a = 6 (i.e., ceiling(sqrt(33)):
    1. b2 = 6 * 6 – 33 = 3
    2. b2 is not a square number
    3. a = a + 1
3. Second iteration: a = 7:
    1. b2 = 7 * 7 – 33 = 16
    2. b2 is a square number
4. Calculate p and q:
    1. p = 7 - sqrt(16) = 3
    2. q = 7 + sqrt(16) = 11

Breaking RSA in Practise

https://arstechnica.com/information-technology/2022/03/researcher-uses-600-year-old-algorithm-to-crack-crypto-keys-found-in-the-wild/

ars TECHNICA

BIZ & IT TECH SCIENCE POLICY CARS GAMING & CULTURE STOR

BREAKING KEYS —

Researcher uses 379-year-old algorithm to crack crypto keys found in the wild

It takes only a second to crack the handful of weak keys. Are there more out there?

DAN GOODIN - 3/14/2022, 9:31 PM

CVE-2022-26320

NIST

NVD MENU

Information Technology Laboratory

NATIONAL VULNERABILITY DATABASE

NIST NATIONAL VULNERABILITY DATABASE NVD

VULNERABILITIES

CVE-2022-26320 Detail

Description

The Rambus SafeZone Basic Crypto Module before 10.4.0, as used in certain Fujifilm (formerly Fuji Xerox) devices before 2022-03-01, Canon imagePROGRAF and imageRUNNER devices through 2022-03-14, and potentially many other devices, generates RSA keys that can be broken with Fermat's factorization method. This allows efficient calculation of private RSA keys from the public key of a TLS certificate.

Severity

CVSS Version 3.x | CVSS Version 2.0

CVSS 3.x Severity and Metrics:

NVD **NIST:** NVD **Base Score:** 9.1 CRITICAL **Vector:** CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N

QUICK INFO

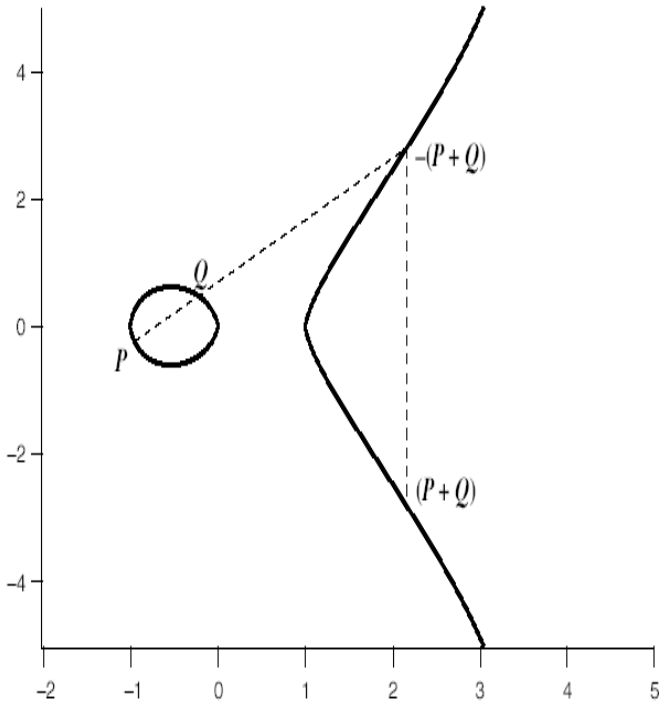
CVE Dictionary Entry:
CVE-2022-26320
NVD Published Date:
03/14/2022
NVD Last Modified:
03/23/2022
Source:
MITRE

Elliptic Curve Cryptography (ECC)

- Traditional methods exploit the properties of arithmetic using large finite groups Z_n with n having a typical size of 1024 bits, i.e. 309 decimal digits
- The security depends on the difficulty of factorising large numbers or calculating discrete logarithms
- Using large numbers makes such algorithms computationally expensive
- In ECC, Z_n is replaced by points of an elliptic curve, making the discrete log calculation problem different and much harder compared to the discrete log in ordinary groups

Elliptic Curve Groups

- Elliptic curves are based on simplified cubic equations, e.g.
  $y^2 = x^3 + ax + b$
  where a and b are real numbers
- The curve shown here is defined by the equation
  $y^2 = x^3 - x$ (i.e., a = -1 and b = 0)
- To plot such a curve, we need to compute
  $y = \mathrm{sqrt}(x^3 + ax + b)$
- Since the shape of the curve depends on a and b, ECs can be described as $E(a, b)$
  - The above curve can be written as $E(-1, 0)$
- In order to operate on elliptic curves, we need to introduce an operation that is equivalent to the addition as well as a "0" element



Elliptic Curves over a Finite Field

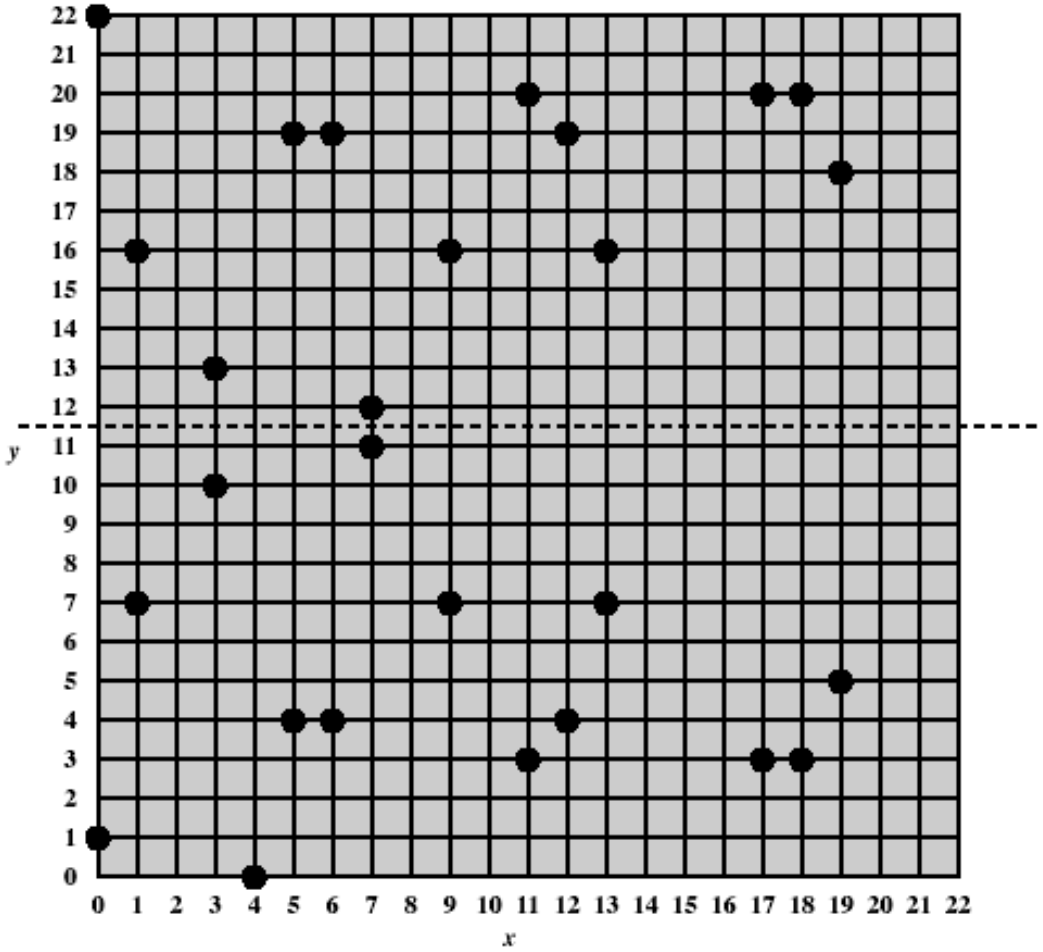
- In order to have values (x, y) within Z_p, the modulus operation is used again:

$$y^2 \bmod p = (x^3 + ax + b) \bmod p$$

- p is either a prime number or $p = 2^m$
- We only consider pairs (x, y), where both x and y are integer values
- Example: Table of all integer solutions for $E_{23}(1,1)$

(0, 1)	(6, 4)	(12, 19)
(0, 22)	(6, 19)	(13, 7)
(1, 7)	(7, 11)	(13, 16)
(1, 16)	(7, 12)	(17, 3)
(3, 10)	(9, 7)	(17, 20)
(3, 13)	(9, 16)	(18, 3)
(4, 0)	(11, 3)	(18, 20)
(5, 4)	(11, 20)	(19, 5)
(5, 19)	(12, 4)	(19, 18)

The Elliptic Curve $E_{23}(1,1)$

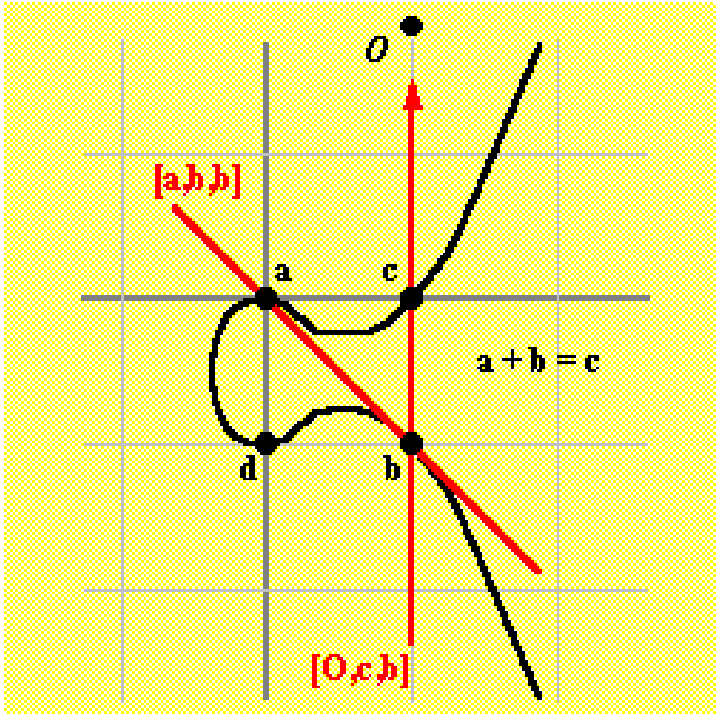


Adding Points on an Elliptic Curve

- ECC requires the equivalent of an addition on $E_p(A,B)$ of two points a and b
- This is done (geometrically) as follows:
  - Draw a straight line through a and b to find the third intersecting point w,
  - then draw a vertical line through w to find the intersecting point c (that's the sum)
- Every line intersects the curve three times (tangents are counted twice), e.g., the line through a and b intersects a "third" point b. We name this line [a,b,b]
- O is called the origin, or point at infinity
- We can say

  a + b = c a + d = b + c = O

  a + a = b a + O = a



ECC over a Finite Field: Addition

- There's p as defined before
- Addition of two field elements $S = (x_S, y_S)$ and $Q = (x_Q, y_Q)$ with $S <> -Q$:
  - $S + Q = R = (x_R, y_R)$
  - $x_R = (L^2 - x_S - x_Q) \bmod p$
  - $y_R = (L (x_S - x_R) - y_S) \bmod p$
  - L is either
    - $((y_Q - y_S) / (x_Q - x_S)) \bmod p$, if $S <> Q$, or
    - $((3 x^2_S + a) / (2 y_S)) \bmod p$, if $S = Q$

ECC over a Finite Field: Addition and Multiplication

- The addition of two elliptic points P and Q consists of a number of integer operations (mod q):
  - 5 or 6 subtractions
  - 1 or 4 multiplications
  - 1 division
- A multiplication $(P * Q)$ is done via consecutive additions
- A scalar multiplication $(x * Q)$ with some scalar x is the operation of successively adding a point Q along an elliptic curve to itself x times (i.e. $Q + Q + Q + \cdots + Q$)

ECC Diffie-Hellman

- Similar to conventional Diffie-Hellman, but operates of finite EC field:
  - Users A & B select a suitable curve $E_p(a, b)$
  - Users select base point (equivalent to primitive root) $G=(x_1, y_1)$
  - User A & B select private keys n_a and n_b
  - Users A & B compute public keys PA and PB
  - Shared keys are exchanged
  - Secret key K is computed

ECC Diffie-Hellman Example

- Use $E_{211}(0, -4)$ that is equivalent to $y^2 \bmod 211 = (x^3 - 4) \bmod 211$
- Choose G = (2, 2)
- User A chooses $n_a = 121$, so A's public key PA is:
  $121 * G = 121 * (2, 2) = (115, 48)$
- User B chooses $n_b = 203$, so B's public key PB is:
  $203 * G = 203 * (2, 2) = (130, 203)$
- The shared secret key K is $121 * (130, 203) = 203 * (115, 48) = (169, 69)$
- Note:
  - ECC-DH (or ECDH for short) can be compromised via a MitM!
  - We still use a BIGNUM integer representation, but the range of values is significantly smaller, and operations can be executed much quicker (see next slide)

Comparable Key Sizes for Equivalent Security

Symmetric scheme (key size in bits)	ECC-based scheme (size of p in bits)	RSA (modulus size in bits)
56	112	512
80	160	1024
112	224	2048
128	256	3072
192	384	7680
256	512	15360

FYI: Curve25519

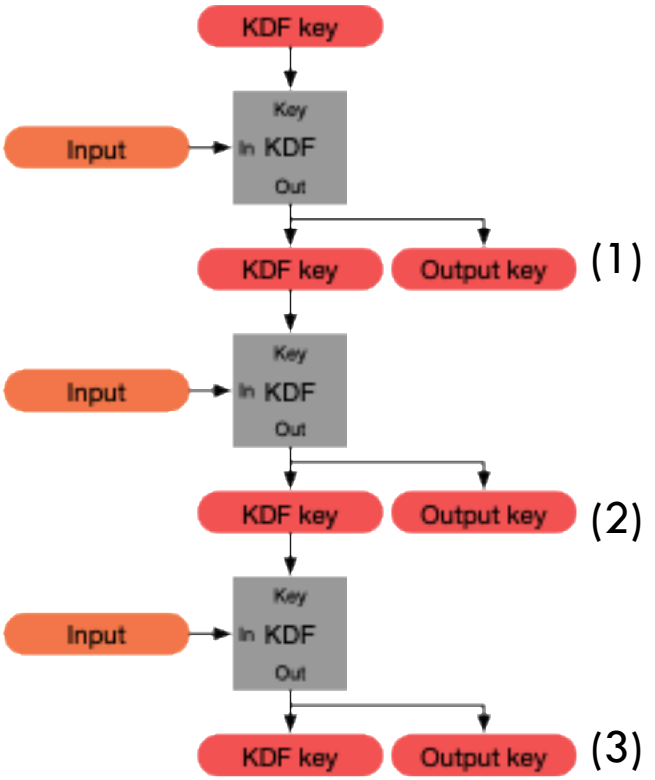
- Curve25519 is an elliptic curve offering 128 bits of security (with 256 bits key size) and designed for use with the elliptic curve Diffie–Hellman (ECDH) key agreement scheme
- It is one of the fastest ECC curves and is not covered by any known patents
- It was first released by the cryptologist Daniel J. Bernstein in 2005
- In 2013, interest began to increase considerably when it was discovered that the NSA had potentially implemented a backdoor into the most common EC encryption method
  - i.e. the P-256 curve based Dual_EC_DRBG algorithm
- Today it is the de facto alternative to P-256
- Its reference implementation is public domain software

The Double-Ratchet Algorithm[1]

- The **Double Ratchet algorithm** is a cryptographic protocol used by two parties to exchange encrypted messages
  - Messages are encrypted using (fast) symmetric key algorithms (e.g., AES)
  - Every message that is exchanged in either direction is encrypted using a different private key
- The algorithm is implemented in the Signal protocol, which in turn is used in secure messaging apps such as the Signal app and WhatsApp
- It ensures forward secrecy and post-compromise security, making conversations secure even if previous keys are compromised
- (Perfect) forward secrecy and post-compromise security are properties of secure communication protocols
  - **Forward security** ensures the confidentiality of past sessions even if long-term keys are compromised
  - **Post-compromise security** ensures the security of future communications even after an initial compromise

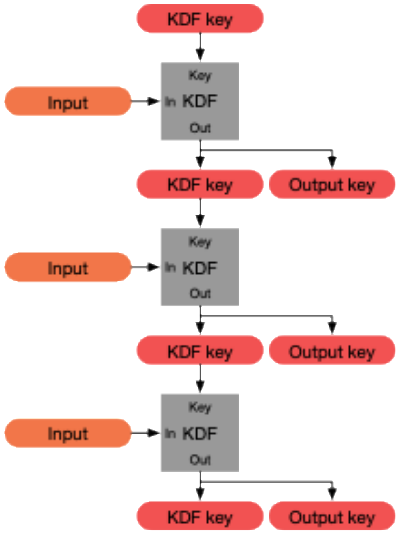
Key Derivation Function (KDF) and KDF Chains

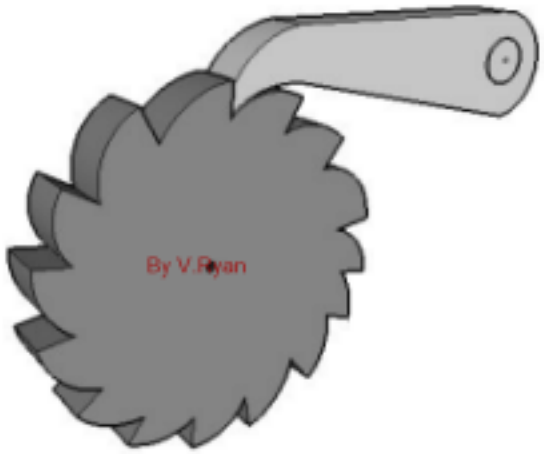
- A **KDF** is a cryptographic function that
  - is used to create a new secret key for each message
  - takes a secret (KDF key) and some (Input) data, and returns an output
  - looks like a "one-way" function (i.e., a hash function)
- In a **KDF chain** some of the KDF output is used as an (Output key) and some is used to make a new (KDF key)
- If two endpoints agree on the same initial (KDF key) and the same (Input), they create the same sequence of output keys, and can exchange messages securely
- A KDF chain guarantees forward security, but not automatically post-compromise security
  - Consider output key (2) being recovered by an attacker:
  - The attacker cannot calculate key (1)
  - The attacker is only prevented from calculating Output key (3), if Input is a secret shared by both endpoints



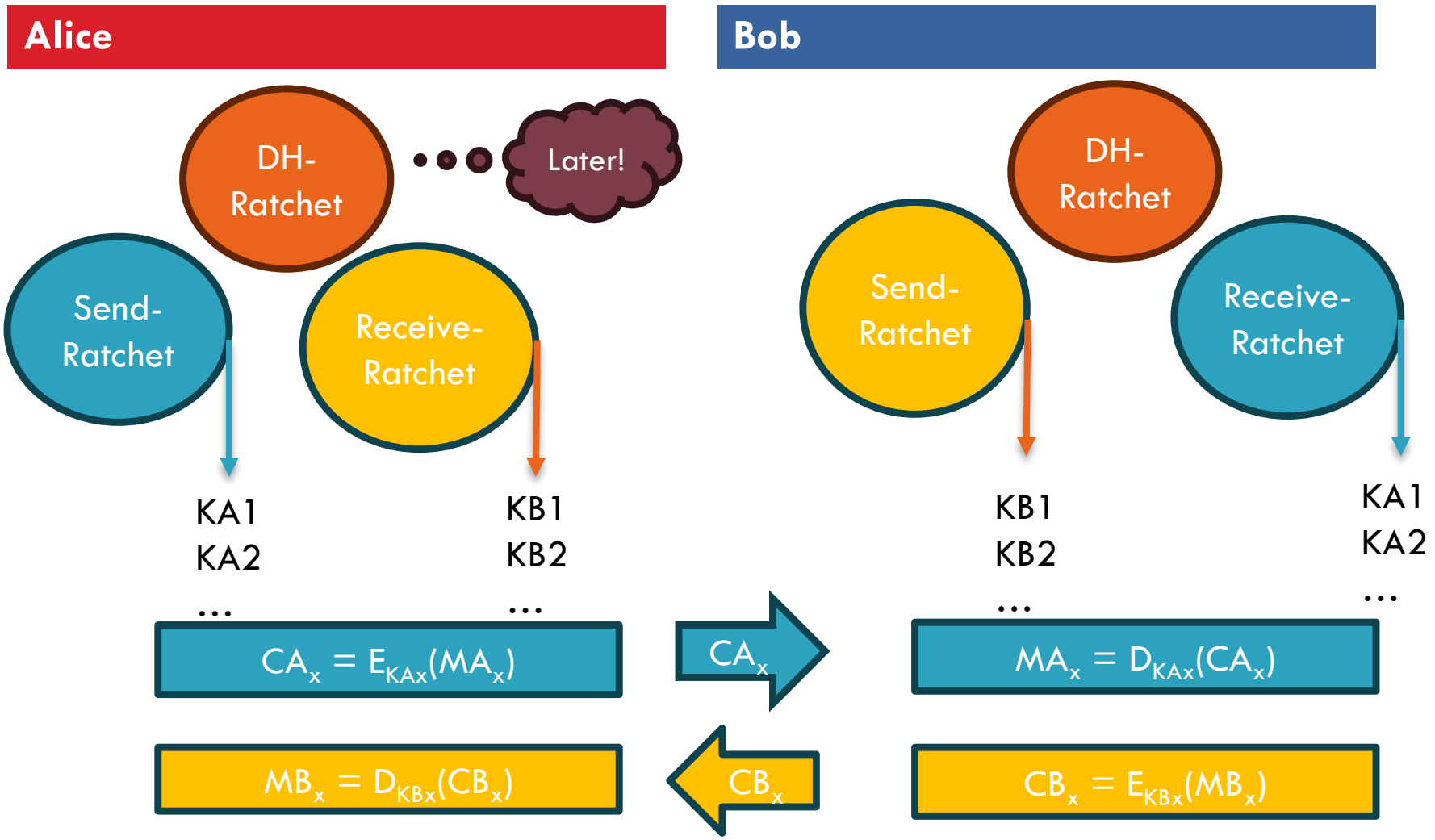
KDF Chains

- A KDF chain is like a ratchet, which only goes in one direction
  - each step provides a different output (KDF key | | Output key)
- Both Alice and Bob have both a "send" and "receive" ratchet each
- Alice's "send" and Bob's "receive" ratchet are initialised using the same initial KDF and the same *Input* key (and visa versa)
- Every time a message is to be sent by either side, it is encrypted first using a new encryption key (*Output key*) that is generated by invoking the KDF (i.e., the "sender" ratchet)
- Similarly, every time the receiver receives a new message it calculates the (same) key for message decryption by invoking the KDF (i.e., the "receiver" ratchet)





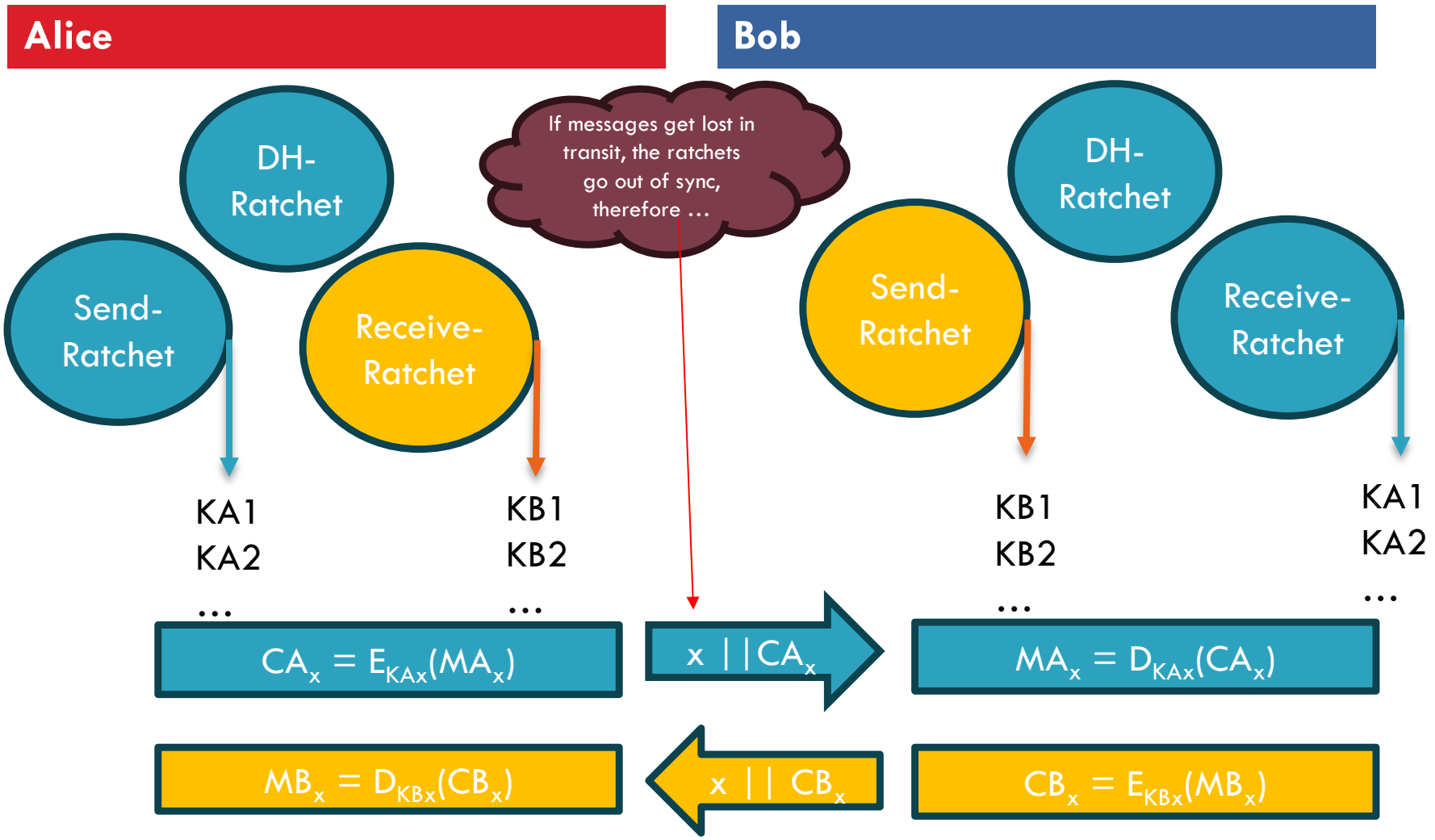
Sender and Receiver Ratchet



Explanations

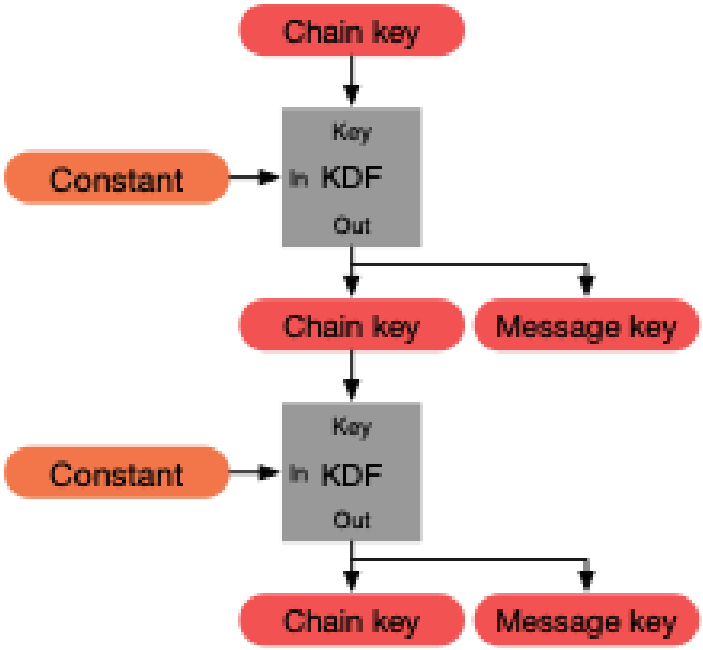
- $K\{A|B\}_x$ is a secret key used by A or B for encoding and decoding a message (e.g., KA_5 or KB_7)
  - x is simply an incremented index value (i.e., 1, 2, 3,…)
- $M\{A|B\}_x$ are (indexed) plaintext messages generated by A or B (e.g., MA_5 or MB_7)
- $C\{A|B\}_x$ is the corresponding ciphertext
  - E.g., MA_3 <-> CA_3
- E() and D() are corresponding encryption and decryption functions that use a key KAx (e.g., $D_{KA5}(CA_5)$)

Synchronising Sender and Receiver Ratchets to compensate for lost Messages



Symmetric Key Ratchet

- "Send" and "receive" ratchets are also called the **symmetric-key ratchets**
- Since every message sent is encrypted with a unique *Message key* (see diagram), the receiver may have to buffer generated (decryption) keys to deal with packets received out-of-order
- Here KDF keys are called *(Chain keys)*
- *The sequence of* generated *chain keys is called* a **sending chain** / **receiving chain**
- Here KDF chains use a (secret) *(Constant)* as a 2nd input to provide post-compromise security



The Diffie-Hellman Ratchet

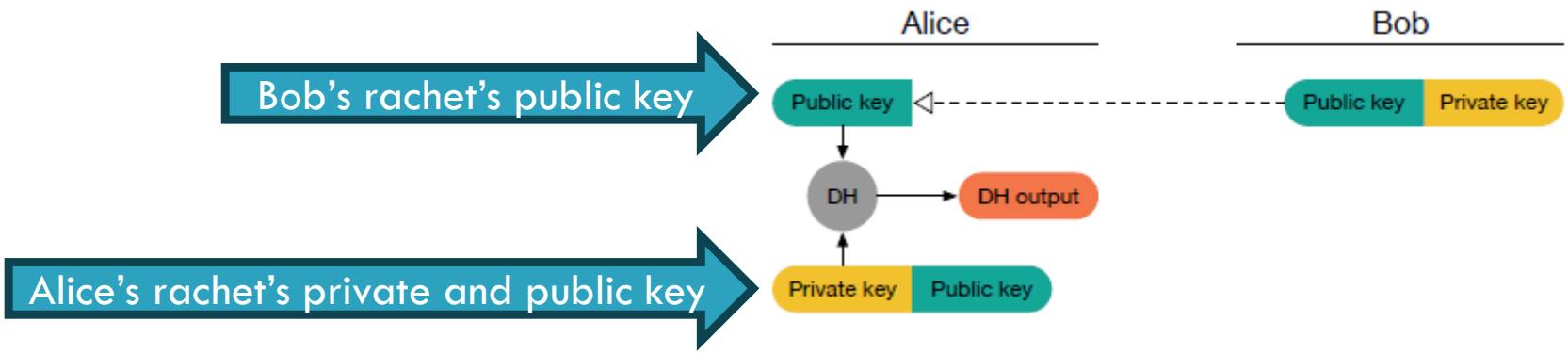
- As Alice and Bob exchange messages, they also exchange new Diffie-Hellman public keys to generate shared secret keys
- These secret keys become the *input* to another KDF chain, the **root chain**
  - This is called the **Diffie-Hellman ratchet**
- The output keys from the root chain provide for new KDF chain keys for the sending and receiving ratchet
- The complete construct is called a Double Ratchet, consisting of the symmetric key ratchets and the DH ratchet, which require KDF keys for three chains:
  - a **sending chain and a receiving chain** (linked to the "send" and "receive "ratchets)
    - With Alice's sending chain matches Bob's receiving chain, and vice versa
  - a **root chain** (linked to the DH-ratchet)

The Diffie-Hellman Ratchet

- To implement the DH ratchet, each party generates a DH key pair (a Diffie-Hellman public key and private key) which becomes their current ratchet key pair
- Every message from either party begins with a header which contains the sender's current DH-ratchet public key
- When a new ratchet public key is received from the other party, a DH ratchet step is performed which replaces the local party's current ratchet key pair with a new key pair
- This results in a "ping-pong" behavior as the parties take turns replacing ratchet key pairs

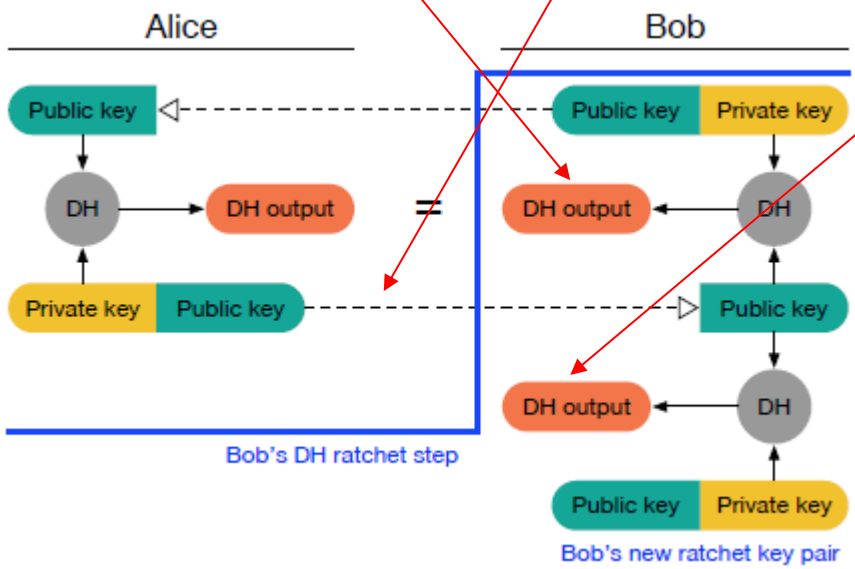
Stepping through the DH-Ratchet: Step 1

- Alice receives Bob's ratchet's public key
  - Alice's ratchet's public key isn't yet known to Bob
- As part of the initialisation Alice performs a DH calculation using her ratchet's (Private key) and Bob's ratchet's (Public key)



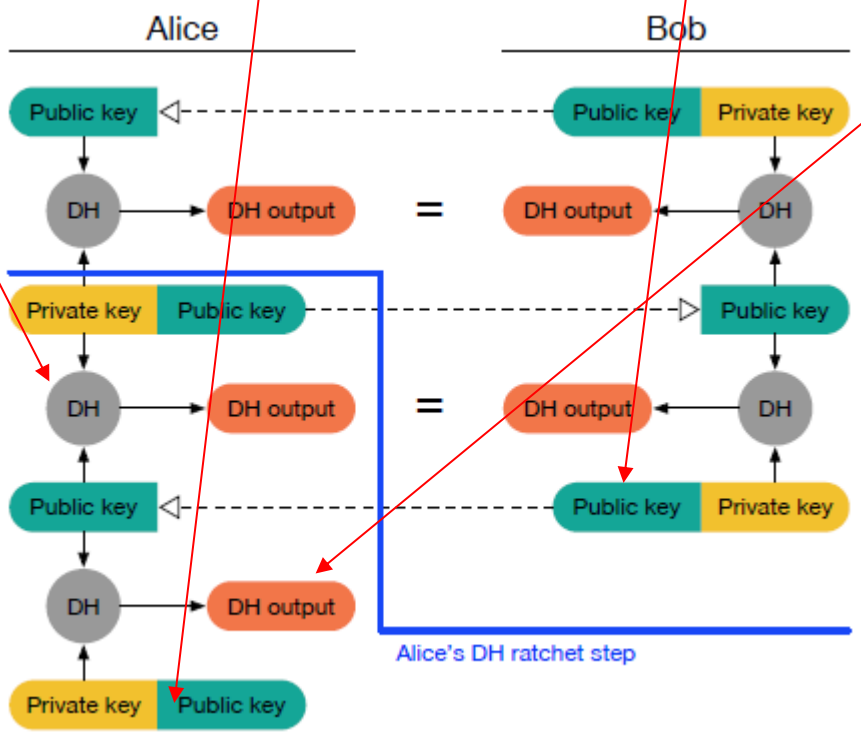
Stepping through the DH-Ratchet: Step 2

- Alice's initial messages advertise her ratchet's public key
- Once Bob receives one of these messages, he performs a DH ratchet step (consisting of two (DH) steps, i.e., Diffie-Hellman key exchange calculations):
  - He calculates the **DH output** between Alice's ratchet's public key and his ratchet's (Private key), which equals Alice's initial (DH output)
  - Bob then calculates a **new ratchet key pair** and calculates a new **DH output**:



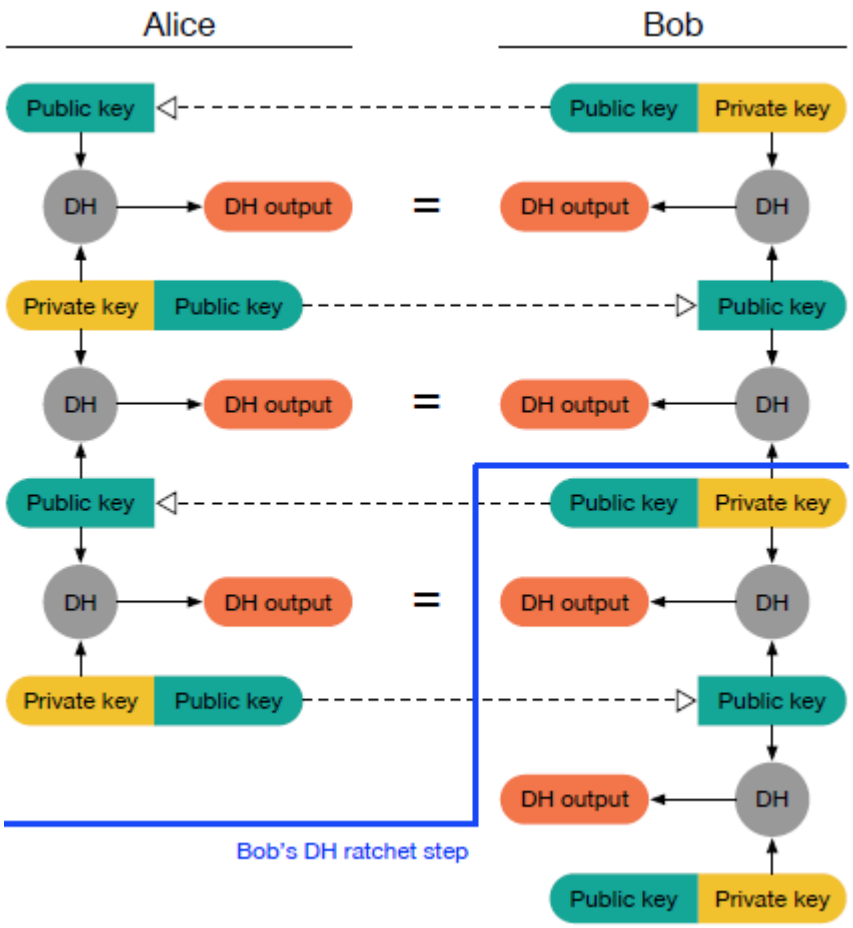
Stepping through the DH-Ratchet: Step 3

- Messages sent by Bob advertising his new Public key are received by Alice, who does a similar step comprising:
  - A (DH) operation using her current Private key and bob's new Public key will result in a DH output identical to the one calculated by Bob
  - She creates a new Private / Public key and calculates a new DH output :



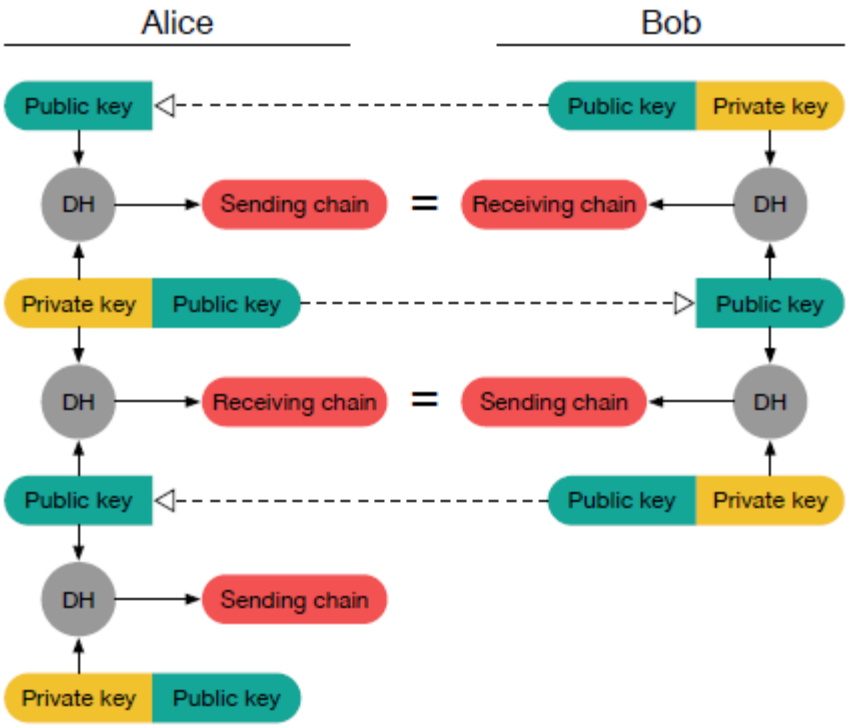
Stepping through the Diffie-Hellman Ratchet: Step 4+

- Messages sent by Alice advertise her new public key
- Bob receives one of these messages and perform a second DH ratchet step, and so on



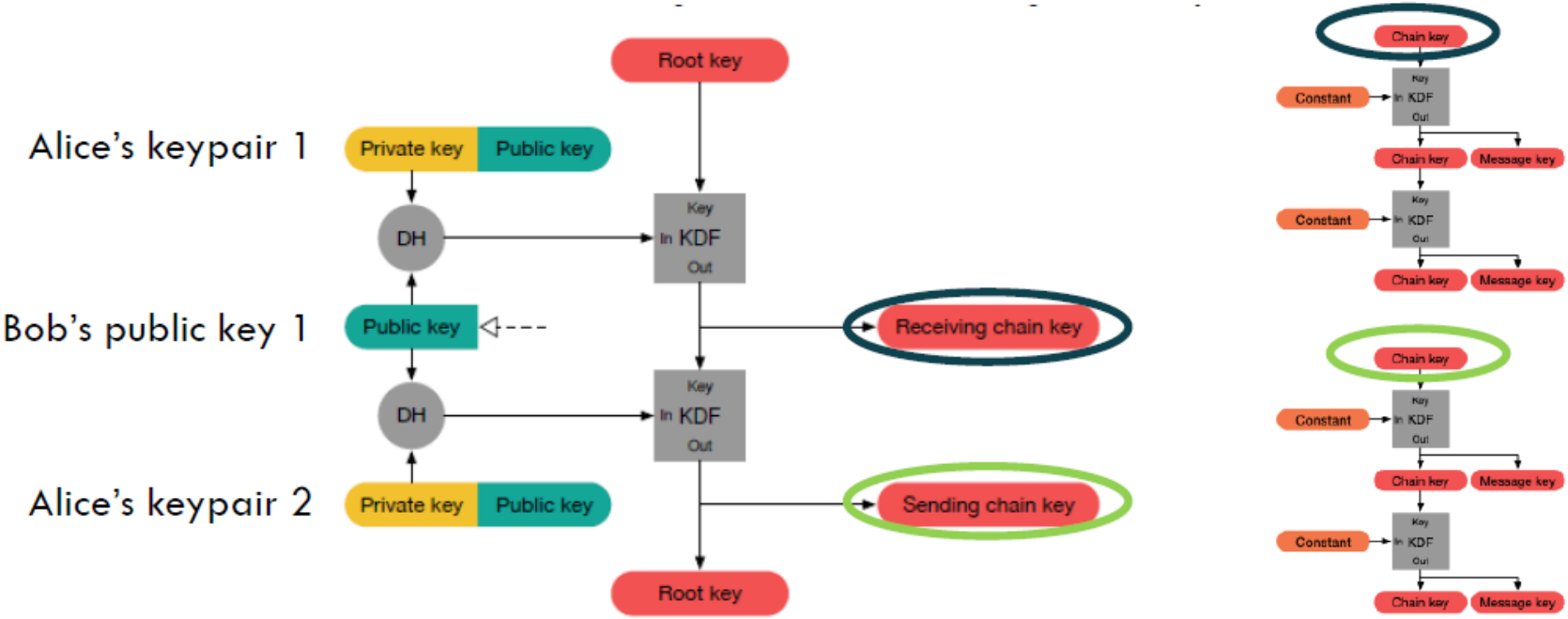
Deriving Sending and Receiving Chains Keys

- The **DH outputs** generated during each DH ratchet step are used to derive new sending and receiving chain keys for Alice's and Bob's symmetric key ratchets
- The DH outputs are not used directly, but go through a DH ratchet first (see next slide)



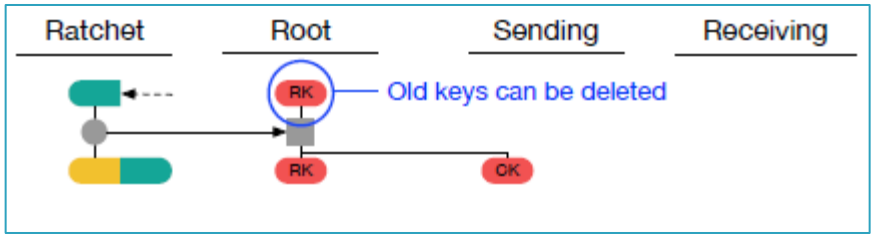
Deriving Sending and Receiving Chains

- This diagram shows the complete process from Alice's perspective:
  - The **Root Key** is a shared secret with Bob, determined via (ECC-) DH at the beginning of the protocol / session
  - The DH output (as calculated in previous slides), together with the Root key, is processed by the DH ratchet in the centre of the diagram to create a *Receiving chain key*
  - Bob's public key, together with Alice's *Private key* of her 2nd generated keypair is used for another KDF invocation that generates the *Sending chain key* and a new *Root key*

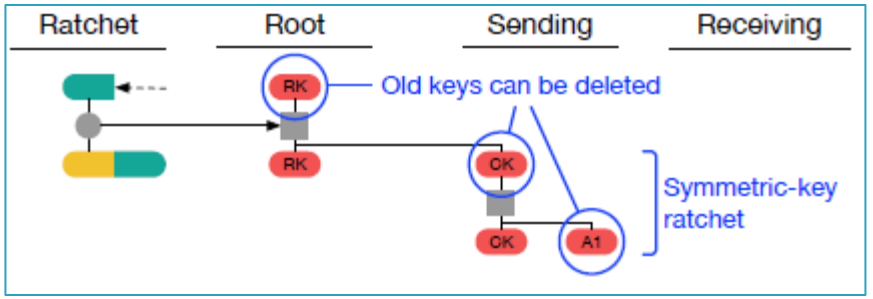


A Double Ratchet Walk-Through

- The following example shows a double ratchet walk-through from Alice's perspective, including only messages she is receiving from Bob
- Step 1:
  - Alice receives Bob's public key and generates a new root key (RK) and sending chain key (CK)

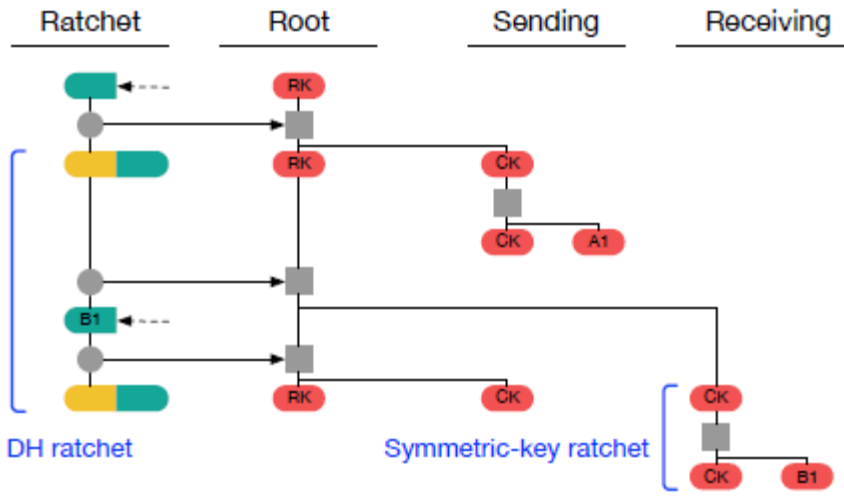


- Step 2:
  - When Alice sends her first message, she applies a symmetric-key ratchet step to her sending chain key (CK), resulting in a
    - message key (A1)
    - new chain key (CK) (ignore for now)



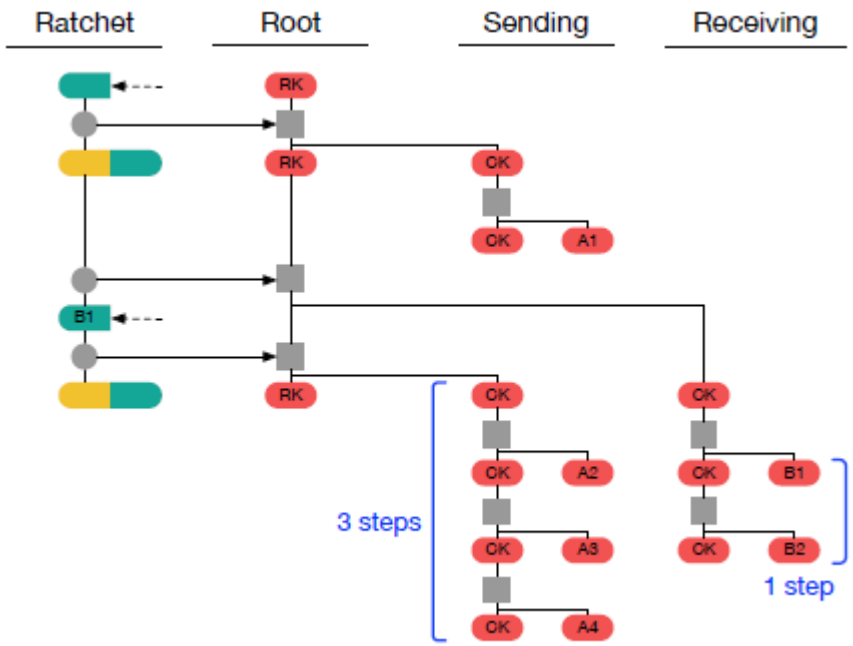
A Double Ratchet Walk-Through

- Step 3:
  - Alice receives a response from Bob; it contains his new DH ratchet public key B1
  - Alice applies a DH ratchet step to derive a new receiving chain key (CK) …
    - She then applies a symmetric-key ratchet step on (CK) to get the message key (B1) for the received message, as well as a new chain key (CK)
  - … and to derive a new sending chain key (CK)
    - In the next step (shown on the next slide), she applies the ratchet on (CK) as well to create the sending key (A2)



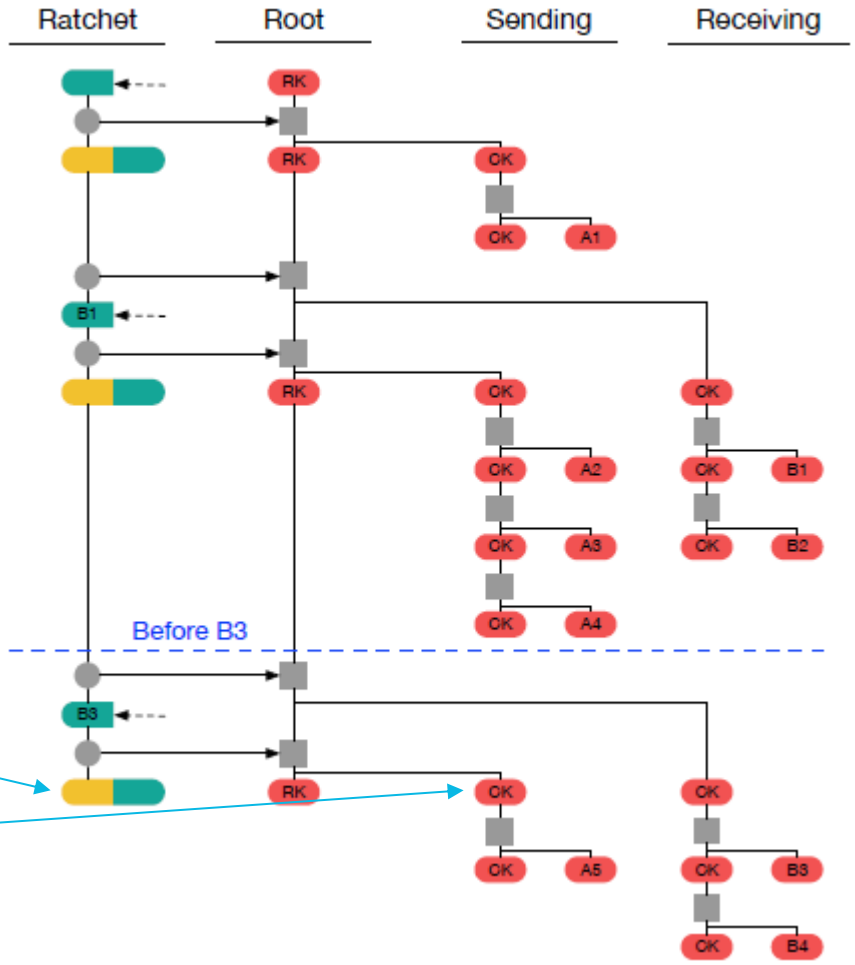
A Double Ratchet Walk-Through

- Step 4:
  - Here Alice next sends a message using (A2), and applies two more ratchet steps to create sending message keys (A3) and (A4) for 2 additional messages
    - Note that the DH-rachet wasn't invoked to create new chain keys, as seen before, i.e. Alice sent a sequence of messages to Bob without prior receiving his new public key
  - Alice receives a message encrypted with (B2)
  - Since Alice didn't receive a new public key from Bob, she simply applies the receiving key ratchet again, to derive (B2)



A Double Ratchet Walk-Through

- Step 5:
  - Alice then receives Bob's new public key (B3), as well as messages encrypted with (B3) and (B4)
  - She generates these keys, by
    - Applying the DH ratchet and creating a new receiving chain key (CK)
    - Executing the receiving key ratchet twice to generate (B3) and (B4)
  - Alice also generates a new sensing message key (A5), by
    - calculating a new private key
    - Applying the DH ratchet
    - Creating a new sending chain key
    - Executing its ratchet once to create (A5)



Summary: Keys and Key Exchanges in the Double Ratchet Protocol

- Initial Key Exchange:
  - Two parties (Alice and Bob) perform an initial key exchange using (a MitM-resilient variation of) ECDH to establish the *Root key;*
  - The Constants in the symmetric key ratchets are derived from the Root key
- Symmetric Key Ratcheting:
  - Every time a message is sent / received, a new symmetric encryption key is provided by the “send” ratchet and the “receive” ratchet
  - This process is known as "ratcheting forward" and ensures that each message has a unique encryption key
- Asymmetric Key Ratcheting:
  - Normally, after each message exchange, both parties generate a new root key by doing a DH key exchange
  - However, if the message receiver is offline, the sender can still use symmetric key ratcheting to create a new message key for each message

References

[1] The Double Ratchet Algorithm; Trevor Perrin and Moxie Marlinspike