# PUBLIC KEY CRYPTOGRAPHY

Applied Cryptography

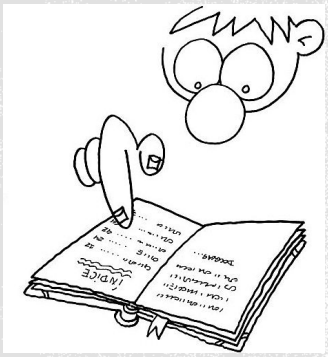
*Silvio Ranise* [ silvio.ranise@unitn.it or *ranise@fbk.eu* ]

UNIVERSITÀ DI TRENTO

FONDAZIONE BRUNO KESSLER

## CONTENTS

- Overview on Public Key Cryptography
  - Confidentiality and Integrity
  - Key exchange
  - Forward Secrecy
- RSA
  - Confidentiality, Integrity and Key Exchange (no Forward Secrecy)
- Diffie-Hellman
  - Key exchange (Forward Secrecy if suitably extended)
- ECC
  - Reducing the key while guaranteeing security

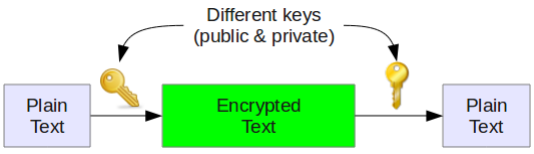


2

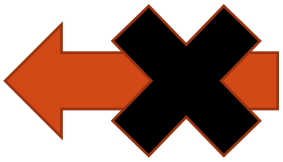
# PUBLIC KEY CRYPTOGRAPHY: INTRODUCTION

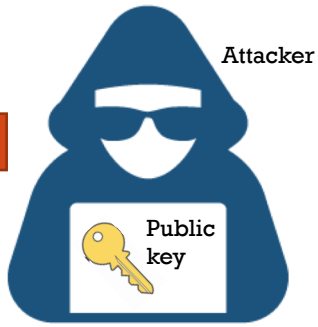


## PUBLIC KEY CRYPTOGRAPHY: OVERVIEW (1)




Deriving a private key from a public key would require to solve a computationally difficult mathematical problem…

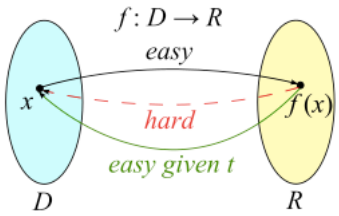





Private key

3

# MATHEMATICAL INTUITION: ONE-WAY TRAPDOOR FUNCTIONS



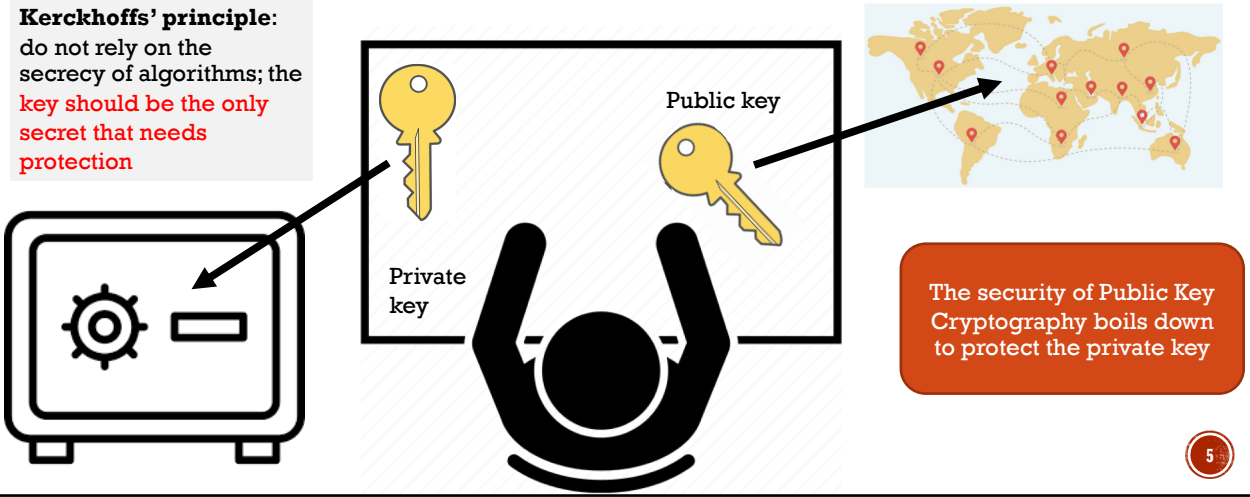

- *Functions* that are *easy to compute whereas their inverse function is difficult to compute* if *a secret t is not known but becomes easy to invert when the secret t is available*
- Examples
  - *Multiplication vs. factorization*
    - Given prime numbers, say 3 and 7, it is easy to calculate the product
    - Given number 21 that is a product of two primes, it is not easy to determine the prime factors
    - Problem becomes much harder if we start with primes that have, say, 400 digits or so, because the product will have ~800 digits
  - *Exponentiation vs. logarithms*
    - Take the number 3 to the 6th power; it is relatively easy to calculate $3^6 = 729$
    - Start with number 729; it is difficult to determine the two integers, $x$ and $y$ s.t. $\log_x 729 = y$
- **With suitable parameters**, these problems are a basis for many cryptographic algorithms
  - *Not all instances of these problems are difficult to solve*

It is unknown if one-way trapdoor functions exist… we believe that certain are indeed so…



# PUBLIC KEY CRYPTOGRAPHY : OVERVIEW (2)

**Kerckhoffs' principle**: do not rely on the secrecy of algorithms; the key should be the only secret that needs protection




The security of Public Key Cryptography boils down to protect the private key

# OVERVIEW

SSH (Secure Shell) is a network protocol that gives users (namely, system administrators) a secure way to access a computer over an unsecured network. It provides strong password authentication and public key authentication, as well as encrypted data communications between two computers connecting over an open network

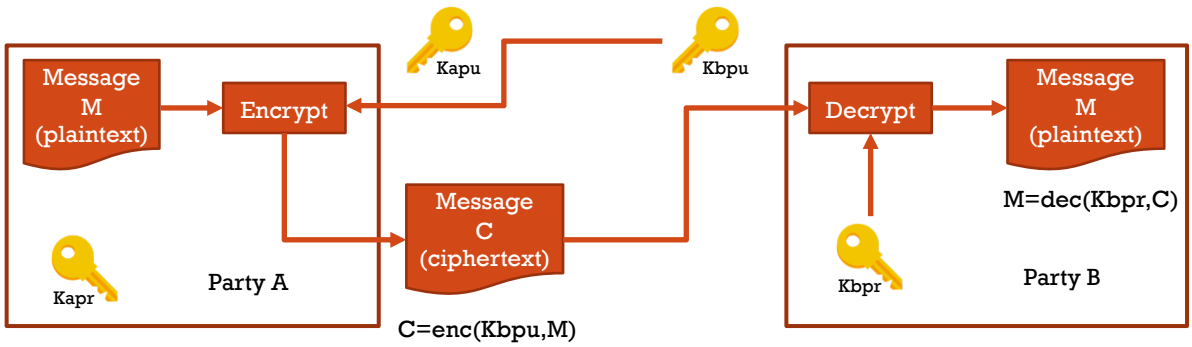
- Public-key cryptography (**PKC**) is also known as **asymmetric-key cryptography**, to distinguish it from the symmetric-key cryptography
- Encryption and decryption are carried out using **two different keys** called
  - **public** key
  - **private** key
- All parties interested in secure communications **publish their public keys**
  - How exactly the public key is made available publicly may vary according to the kind of applications
  - **Example 1**: SSH makes available a public key by storing it in a particular location and allowing other machines to access it trough a given port (22)
  - **Example 2**: SSL/TLS uses X.509 certificates that are exchanged between clients and servers during handshakes
- This side-steps the problem of key distribution that plagued symmetric key cryptography



# SECURITY

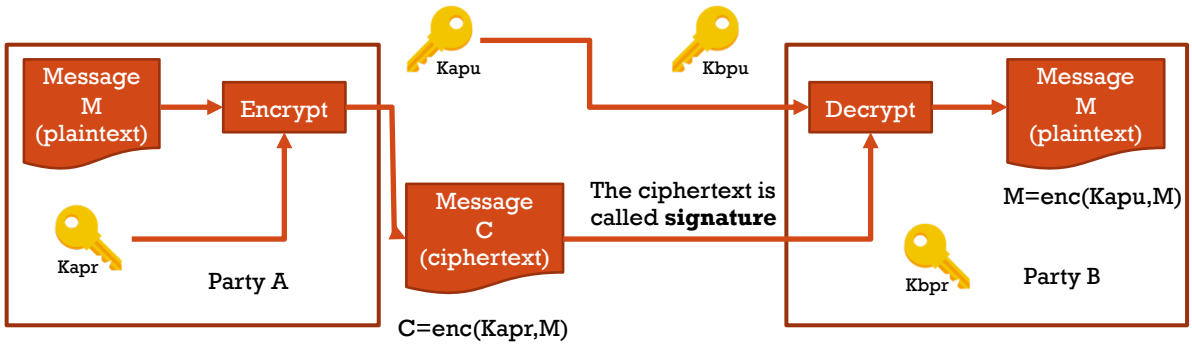
- Secure communication between party A and party B
- Secure = **confidential**
  - Party A can encrypt a message using B's publicly available key
  - Such a message would only be decipherable by B under the assumption that only B have access to the corresponding private key
- Secure = **authentic**
  - Party A can encrypt the message with A's own private key
  - Since this message would only be decipherable with A's public key, that would establish the authenticity of the message, i.e. A is indeed the source of the message
- It is possible to combine the two techniques and provide both confidentiality and authenticity for any message exchanged between A and B

# PRESERVING CONFIDENTIALITY






# PRESERVING AUTHENTICITY






**Note**: in reality, it is not the whole message to be encrypted but rather a digest produced by a hash function

# PRESERVING BOTH CONFIDENTIALITY AND AUTHENTICITY

- **Sender: party A**

1. C1=enc(Kapr,M)
   - Encrypt with its private key plaintext so to enable the check of authenticity
2. C2=enc(Kbpu,C1)
   - Encrypt with public key of receiver so to enable the check for confidentiality

- **Receiver: party B**

1. M'=dec(Kbpr,C2)
   - Decrypt with its private key so to check for confidentiality: notice that M'=C1
2. M''=dec(Kapu,M')
   - Decrypt with sender private key to check for authenticity: notice that M''=M



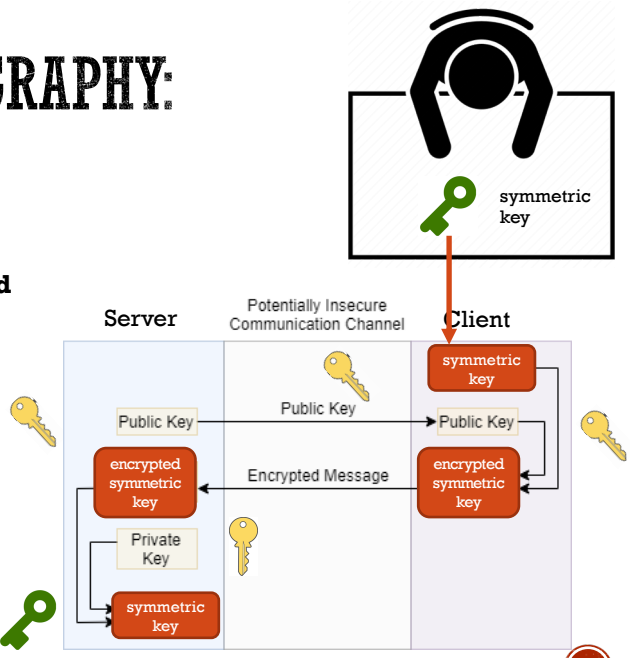
# REMARKS

- The price to pay for achieving both confidentiality and authentication is that a **message must be processed 4 times** per exchange
  - 2 encryptions at the sender's place and 2 decryptions at the receiver's place
  - each of these four steps involves the cost of the public-key algorithm
- PKC does not make obsolete symmetric-key cryptography
  - Because of the greater computational cost associated with PKC…
  - … symmetric-key systems continue to be widely used for content encryption
- PKC has proved indispensable for key management, for distributing the keys needed for the more traditional symmetric key cryptography, for digital signature applications, …

# PUBLIC KEY CRYPTOGRAPHY: KEY EXCHANGE (1)

Public key cryptography: **RSA**

- 1977: **used in almost all Internet-based commercial transactions**
- Not only secure key distribution over insecure channel

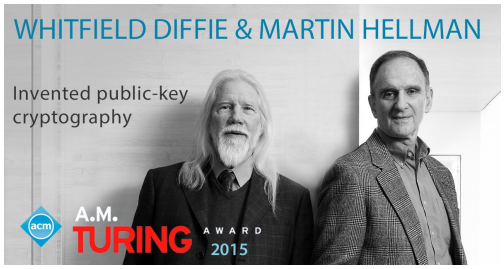



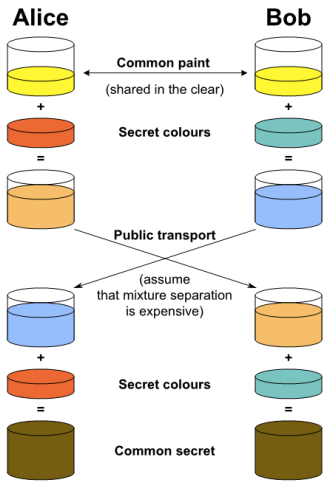

12

# PUBLIC KEY CRYPTOGRAPHY: KEY EXCHANGE (2)

Public key cryptography: **Diffie-Hellman**

- 1976: most significant result in cryptography in the last 300-400 year
- *Secure key distribution over insecure channel*



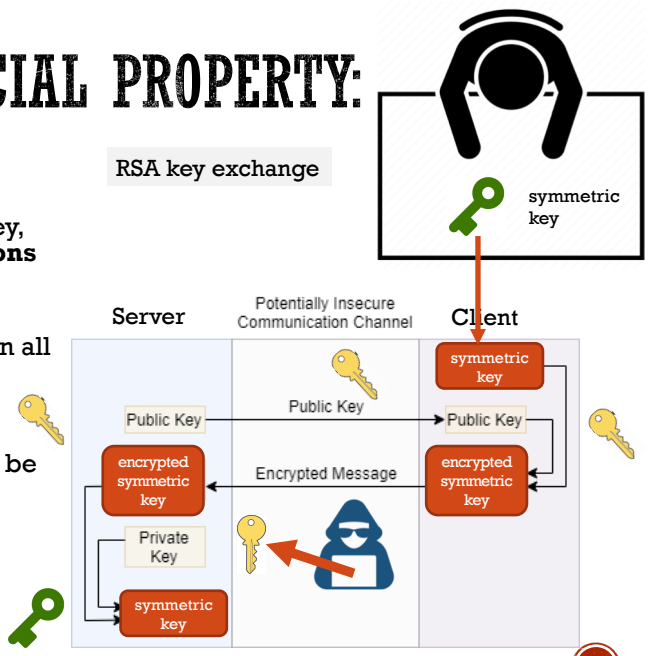




13

# KEY EXCHANGE CRUCIAL PROPERTY: FORWARD SECRECY

- If someone steals the server's private key, then all **past and future communications** are **compromised with all clients**
- This is so because if attackers have recorded past encrypted messages, then all past and future symmetric keys can be decrypted
- **(Perfect) forward secrecy** gives assurances that symmetric keys will not be compromised even if private keys are compromised
- *Extensions of the Diffie-Hellman key exchange support forward secrecy*

RSA key exchange




14

# TWO WAYS TO OBTAIN FORWARD SECRECY

1. A **temporary private key** is generated **for each session**
   - Every session has a different set of keys
   - It is impossible to decrypt past traffic without having complete access to the server at the time-zero
2. Use **ephemeral key exchange**
   - It requires generating a **new public/private key pair per session**
   - It is impossible to decrypt past communications even if the private key of the involved parties is compromised later

To understand why Forward Secrecy is crucial to avoid mass surveillance as desired by certain Legal Enforcement Agencies around the world, you can read the following blog post: https://www.computerworld.com/article/2473792/perfect-forward-secrecy-can-block-the-nsa-from-secure-web-pages--but-no-one-uses-it.html



15

# FORWARD SECRECY, KEY SIZES, AND AN EVOLVING THREAT LANDSCAPE

- The size of RSA keys has increased over time because of the ease of accessing more and more computational power that can be used to derive information about private keys from public keys…

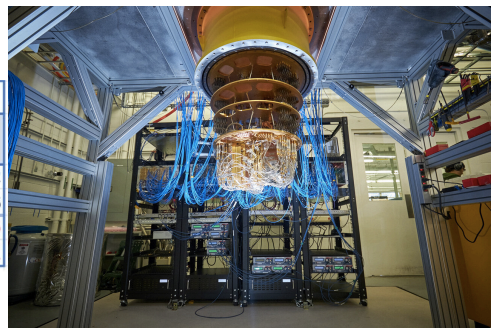
| | Minimum size (bits) of Public Keys | | | Key Size Ratio | Protection from |
|---|---|---|---|---|---|
| Security (bits) | DSA | RSA | ECC | ECC to RSA/DSA | Attack |
| 80 | 1024 | 1024 | 160-223 | 1:6 | Until 2010 |
| 112 | 2048 | 2048 | 224-255 | 1:9 | Until 2030 |
| 128 | 3072 | 3072 | 256-383 | 1:12 | Beyond 2031 |
| 192 | 7680 | 7680 | 384-511 | 1:20 | |
| 256 | 15360 | 15360 | 512+ | 1:30 | |

*Table 1: National Institute of Standards and Technology Guidelines for Public-Key Sizes*

And the situation may get worse if quantum computing becomes available…

16

| Type of Attack | Symmetric Encryption | | | Public Key Encryption | | |
|---|---|---|---|---|---|---|
| | | Key Length | Bits of Security | | Key Length | Bits of Security |
| Classical Computers | AES-128 | 128 | 128 | RSA-2048 | 2048 | 112 |
| | AES-256 | 256 | 256 | RSA-15360 | 15,360 | 256 |
| Quantum Computers | AES-128 | 128 | 64 | RSA-2048 | 2048 | 25 |
| | AES-256 | 256 | 128 | RSA-15360 | 15,360 | 31 |



# WHAT ABOUT THE ADVENT OF QUANTUM COMPUTERS?

17

https://www.technologyreview.com/2021/11/03/1039171/hackers-quantum-computers-us-homeland-security-cryptography/

MIT Technology Review

Featured Topics Newsletters Events Podcasts Sign in Subscribe

COMPUTING

# The US is worried that hackers are stealing data today so quantum computers can crack it in a decade

The US government is starting a generation-long battle against the threat next-generation computers pose to encryption.

By Patrick Howell O'Neill November 3, 2021

19

# RSA
# RIVEST, SHAMIR, ADLEMAN

Based on multiplication vs factorization into primes

# OVERVIEW

- The RSA method was published in scientific paper in 1977
  - The authors were inspired by 1976 seminal paper by Diffie and Hellman
- Historically, the PKC implemented by the RSA algorithm was foundational to the e-commerce revolution in the 1980's
- An RSA user creates and publishes a **public key based on two large prime numbers**
  - The prime numbers are kept secret
  - Messages can be encrypted by anyone, via the public key, but can only be decoded by someone who knows the prime numbers
- **The security of RSA relies on the practical difficulty of factoring the product of two large prime numbers, the "factoring problem"**
  - Breaking RSA encryption is known as the RSA problem
  - Whether it is as difficult as the factoring problem is an open question
- There are no published methods to defeat the system if a large enough key is used
  - Notice that large means really large: **an 829-bit key (integer with 250 decimal digits) has been broken**
  - For details about the RSA numbers and RSA challenge (ended in 2007), have a look at
    - https://en.wikipedia.org/wiki/RSA_numbers



# THE 829-BIT KEY AND ITS FACTORS

```
RSA-250 = 2140324650240744961264423072839333563008614715144755017797754920881418023447
          1401366433455190958046796109928518724709145876873962619215573630474547705208
          0511905649310668769159001975940569345745223058932597669747168173806936489469
          987157849497593749793 7
```

```
RSA-250 = 6413528947707158027879019017057738908482501474294344720811685963202453234463
          0238623598752668347708737661925585694639798853367
        × 3337202759497815655622601060535511422794076034476755466678452098702384172921
          0037080257448673296881877565718986258036932062711
```

https://lists.gforge.inria.fr/pipermail/cado-nfs-discuss/2020-February/001166.html

# RSA: MAIN IDEA

- Starting point: **Euler's Theorem**
  - for every positive integer $n$ and every $a$ that is **coprime** to $n$, the following must be true

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

  where $\phi(n)$ is the **totient** of $n$

- Notice that Euler's totient function $\phi(n)$ counts the positive integers up to a given integer $n$ that are relatively prime to or, equivalently, coprime with $n$



# AN APPLICATION OF EULER THEOREM

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

- It is possible to use Euler theorem to greatly simplify the computations required to compute large powers modulo $n$
- Example
  - Consider computing $7^{222}\ (mod\ 10)$
  - Computing first $7^{222}$ gives a number larger than $4 * 10^{187}$ (this is not very easy to do)
  - Instead, observe that 7 and 10 are coprime and that $\phi(10) = 4$
  - Because of Euler theorem, we have $7^4 \equiv 1\ (mod\ 10)$
  - Thus, we can derive $7^{222} \equiv 7^{4*55+2} \equiv (7^4)^{55} * 7^2 \equiv (1)^{55} * 7^2 \equiv 49 \equiv 9\ (mod\ 10)$

- In general, one can show that if $x \equiv y \pmod{\varphi(n)}$, then $a^x \equiv a^y \pmod{n}$

24

# END OF DIGRESSION ON EULER THEOREM



# RSA: MAIN IDEA

Recall that

if $x \equiv y \pmod{\varphi(n)}$, then $a^x \equiv a^y \pmod{n}$

- *Application*
  - Let *M* be is an integer that represents a message
    - Any bit string represents some integer, no matter how large
  - Pick two integers *e* and *d* such that $e * d \equiv 1 \ (mod\ \phi(n))$
  - **Assume that *M* is coprime with the modulus *n***
  - Then, we have $M^{e*d} \equiv M^{e*d \bmod \phi(n)} \equiv M \ (mod\ n)$
- The assumption that *M* and the modulus *n* are coprime can be weakened by requiring that **the modulus *n* is the product of two primes**
- In fact, under this hypothesis, **the last congruence relation above holds for any value of the message *M* between 0 and *n-1*** (both included)
- This leads to the core of the RSA encryption algorithm…

# RSA: CORE ALGORITHMS (1)

Preconditions

| Parameter | Definition |
|---|---|
| $n$ | a modulus for modular arithmetic |
| $\phi(n)$ | the totient of $n$ |
| $e$ | an integer that is coprime with $\phi(n)$ |
| $d$ | an integer that is the multiplicative inverse of $e$ modulo $\phi(n)$ |

This guarantees the existence of the multiplicative inverse modulo $\phi(n)$



# RSA: CORE ALGORITHMS (2)

- **Encryption**
  - *Input*: a plaintext message *M* encoded as an integer between 0 and *n-1* (included)
  - *Output*: a ciphertext $C = M^e \ mod \ n$
- **Decryption**
  - *Input*: a ciphertext *C* obtained as explained above
  - *Output*: a plaintext $M = C^d \ mod \ n$
- **Observation**

$$C^d \ mod \ n = (M^e \ mod \ n)^d \ mod \ n = M^{e*d \ mod \ \phi(n)} \ mod \ n = M \ mod \ n = M$$

# RSA FOR CONFIDENTIALITY (1)

- An entity A who wishes to receive messages confidentially will use
  - the pair of integers $\{e, n\}$ as its **public** key
  - the pair of integers $\{d, n\}$ as the **private** key
  - The parameters $n$, $e$, and $d$ are as introduced above
- Another entity B wishing to send a message $M$ to A confidentially will encrypt $M$ using A's public key $\{e, n\}$ to create the ciphertext $C$
- Only A will be able to decrypt $C$ using its private key $\{d, n\}$
- If the plaintext message $M$ is too long, B may choose to use RSA as a block cipher for encrypting the message meant for A
  - When RSA is used as a block cipher, the block size is likely to be half the number of bits required to represent the modulus $n$



# RSA FOR CONFIDENTIALITY (2)

- *In theory*, **RSA can be used as a block cipher**
- *In practice*, because of its high computational overhead, **RSA is more likely to be used just for server authentication and for exchanging a secret session key**
- Subsequently, a session key generated with the help of RSA-based encryption can be used for content encryption using symmetric-key cryptography

- **QUESTION**
  - What conditions (if any) must be satisfied by the modulus $n$ for the encryption/decryption schema to work in practice?
  - In other words: **how should we choose the modulus $n$ for the RSA algorithm to be considered secure?**

# HOW TO CHOOSE THE MODULUS?

- Recall that

$$M^{e*d} \equiv M^{e*d \bmod \phi(n)} \equiv M \ (mod\ n)$$

under the assumption that ***M* is coprime with the modulus *n***

- It was shown by Rivest, Shamir, and Adleman that **the same property holds for all *M* if the modulus *n* is a product of two prime numbers**, i.e.

$$n = p \times q \text{ for some prime } p \text{ and prime } q$$

- This is necessary to prove the correctness of the RSA algorithm based on the following two key properties…



# PRACTICAL REMARKS

- It is crucial that both $p$ and q be **very large primes** so that also $n=p*q$ is also so
  - Finding the prime factors of a large integer $n$ is computationally harder than determining its primality
  - In this way, the cipher cannot be broken by an exhaustive search for the prime factors of the modulus $n$
- We also need to ensure that $n$ is **not factorizable by one of the modern integer factorization algorithms**
  - More on this below

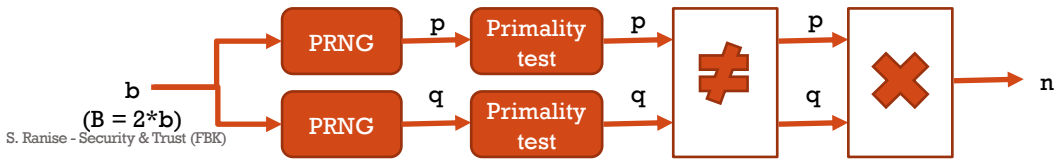
32

# COMPUTATIONAL ASPECTS OF KEY GENERATION IN THE RSA ALGORITHM



## KEY GENERATION

1. ***Generate two distinct primes p and q*** How can we generate primes?
2. Calculate the modulus $n = p * q$
3. Calculate the totient $\phi(n) = (p - 1) * (q - 1)$
4. Select for **public exponent** an integer $e$ such that $1<e<\phi(n)$ and $gcd(\phi(n), e) = 1$
5. Calculate for the **private exponent** a value $d$ such that $d = e^{-1} \bmod \phi(n)$
6. Set public key to {e, n}
7. Set private key to {d, n}



34

# DIGRESSION ON PRNG AND PRIMALITY TESTING



## GENERAL REMARKS ON PSEUDO RANDOM NUMBER GENERATORS

- Widely used in a variety of different applications
  - The Key Distribution Center (KDC) in Kerberos needs to generate a symmetric key for securing the session between two entities; this can be done by using a sequence of random numbers representing bits (e.g., 32 random integers, each one represented as byte can be concatenated to form a 128 bits keys for AES)
  - Nonces used in cryptographic protocols can be seen as random integers
  - As we will see in few slides, random numbers are useful also to generate the compute the modulus in RSA

- What is it a **random number generator**?
  - Simple answer: **a device that generates sequences of numbers that look random**

# ON RANDOMNESS

- To be considered truly random, a sequence of numbers must exhibit the following two properties:
  1. **Uniform Distribution**
     - All the numbers in a designated range must occur equally often
  2. **Independence**
     - Even if one knows some or all the numbers up to a certain point in a random sequence, he/she should not be able to predict the next one (or any of the future ones)
- Truly random numbers can only be generated by physical phenomena
  - Examples: thermal noise, cards, dice, …
- Computers try to **approximate** (simulate) truly random numbers through a variety of approaches



# PSEUDO RANDOMNESS

Are such sequences **cryptographically secure**? I.e. how difficult is it for an attacker to predict the next number from the numbers already in his/her possession?

- Sequence of numbers that are **algorithmically generated**
- Despite being approximations of random sequences of numbers, the **repeatability** of pseudorandom numbers is of great importance in several applications and in particular for cryptography
- Several different approaches that approximate random sequences with different levels of precision
- **Linear congruential generators**

$$X_{n+1} = (a \cdot X_n + c) \bmod m$$

$X_0$ is the seed

Under suitable conditions, it is possible to obtain sequences that starts repeating after *m* randomly generated numbers

$$X_{n+1} = (7^5 \cdot X_n) \bmod (2^{31} - 1)$$

Statistical tests for uniformity and independence show that sequences generated with this recurrence are statistically indistinguishable from true random sequences consisting of positive integers greater than 0 and less than *m*

# ON CRYPTOGRAPHICALLY SECURE PRNGS

The **Linux kernel** generates entropy from keyboard timings, mouse movements, and IDE timings and makes the random data available to other operating system processes through the special files /dev/random and /dev/urandom

- It is a PRNG with properties that make it suitable for use in cryptographic applications including the generation of nonces or keys
- Applications have different requirements: for the generation of
  - nonces: requires **uniqueness**
  - keys: requires **entropy** (i.e. randomness collected by an operating system often collected from hardware sources such as variance in fan noise or HDD)
- Cryptographically secure PRNGs should hold up well under serious attacks, even when part of their initial or running state becomes available to an attacker

More details on weaknesses of cryptographically secure PRNGs can be found in https://link.springer.com/content/pdf/10.1007/3-540-69710-1_12.pdf



# CRYPTOGRAPHICALLY SECURE PRNG?

- Linear congruential generators are not cryptographically secure
- Since it is possible to recover the values of a, c, and m by solving 3 congruences

$$
\begin{aligned}
X_1 &= (a \cdot X_0 + c) \bmod m \\
X_2 &= (a \cdot X_1 + c) \bmod m \\
X_3 &= (a \cdot X_2 + c) \bmod m
\end{aligned}
$$

- As a consequence, **even when a PRNG produces a random looking sequence, it may not be secure enough for cryptographic applications**
- A pseudorandom sequence produced by a PRNG can be made more cryptographic secure by **restarting the sequence with a different seed after every N numbers**
  - One way to do this would be to take the current clock time modulo *m* as a new seed after a given numbers of the sequence have been produced

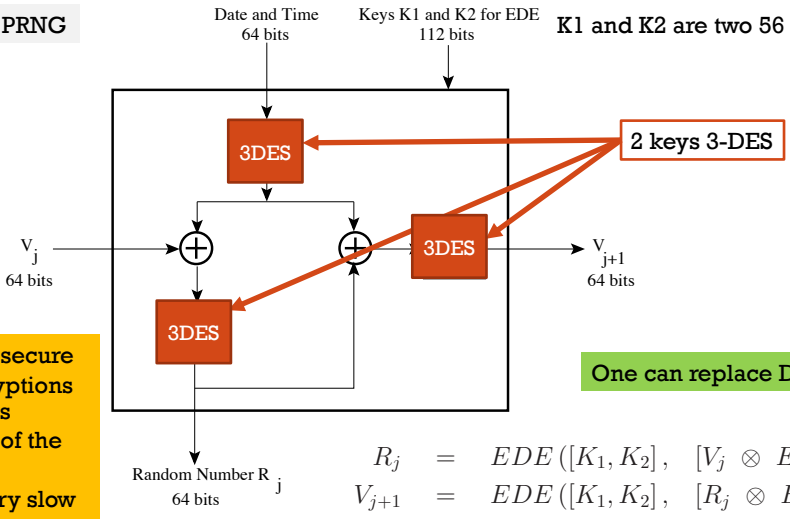
# OTHER EXAMPLES OF CRYPTOGRAPHICALLY INSECURE PRNGS

- Linear Feedback Shift Registers
  - Recall that it is possible to perform attacks based on Linear Algebra
  - Possible mitigations: combine with non-linear components… it should be done with ingenuity and care
- Array shuffling algorithm underlying RC4
  - Weaknesses already at the statistical level, i.e. sequences generated do not really look like random
  - No mitigations are possible… RC4 deprecated as a consequence



# A CRYPTOGRAPHICALLY SECURE PRNG

ANSI X9.17/X9.31 PRNG




Cryptographically secure because DES encryptions makes relationships between elements of the sequences very challenging but very slow

One can replace DES with AES…

$$R_j = EDE([K_1, K_2], [V_j \otimes EDE([K_1, K_2], DT_j)])$$
$$V_{j+1} = EDE([K_1, K_2], [R_j \otimes EDE([K_1, K_2], DT_j)])$$

# GENERAL REMARKS ON PRIMALITY TESTING

- Primality testing is much easier than factoring
- Indeed, to check primality there is no need to factor the number under consideration
- There are two types of primality testing algorithms
  - **Deterministic tests** determine with absolute certainty whether a number is prime
  - **Probabilistic** tests can potentially (although with very small probability) falsely identify a composite number as prime (although not vice versa)
    - In general much faster than deterministic tests
    - Numbers that have passed a probabilistic prime test are therefore properly referred to as probable primes until their primality can be demonstrated deterministically



# MILLER-RABIN ALGORITHM (1)

- Recall Fermat's Little Theorem

If $p$ is prime, then for any integer $a$ that is coprime with $p$, we have $a^{p-1} \equiv 1 \ (mod\ p)$

Or, equivalently,

If $p$ is prime, then for any integer $a$ that is coprime with $p$, we have that $a^{p-1} - 1$ is divisible by $p$

- It is possible to use this result to design a probabilistic algorithm for primality testing
- Let $n$ be the integer that we would like to check for primality
- Randomly select an integer $a$
- if $a^{n-1}\ mod\ n$ **does not** yield 1, then $n$ is not a prime; otherwise, you cannot be certain…

# MILLER-RABIN ALGORITHM (2)

- If $a^{n-1} \bmod n$ **does** yield 1, then $n$ may be a prime or not
  - Example: for $n=25$ and $a=7$, we have $a^{n-1} \bmod n = 7^{25-1} \bmod 25 = 1$
- Since Fermat's Little Theorem must hold for any integer $a$ coprime with $n$, we can select another value…
  - Example: for $n=25$ and $a=11$, we have $a^{n-1} \bmod n = 11^{25-1} \bmod 25 = 16$ which is different from 1 and allows us to conclude that 25 is not a prime
- In other words, we can consider different **probes**, i.e. values for $a$
- Intuitively, the larger the number of probes (i.e. values for $a$), the greater the probability that $n$ is prime (provided that $a^{n-1} \bmod n$ **does not** yield 1 for each value of $a$)
- The tests can be stopped as soon as $a^{n-1} \bmod n$ **does not** yield 1 as we are guaranteed that $n$ is not prime



# MILLER-RABIN ALGORITHM (3)

It exists also a non-probabilistic primality test based on a generalization of Fermat's Little Theorem with much higher computational complexity

- This is the core idea underlying Miller-Rabin algorithm for primality testing
- Implementations of the algorithm exploits several additional properties
  - Example: for integers less than 341,550,071,728,321, one can consider only the following probes 2, 3, 5, 7, 11, 13, and 17
  - Notice that primes used in RSA are usually much larger than the value above
- It is possible to show that the Miller-Rabin algorithm will return `prime' for a non-prime number $n$ with a probability of $4^{-t}$ where $t$ is the number of probes used
- In practice, the probability of an erroneous classification of the input integer is significantly less than $4^{-t}$ provided that certain implementation best practices are adopted
- The computational complexity of the algorithm is $O(t * (log(n))^3)$ where $n$ is the integer being tested

46

# END OF DIGRESSION ON PRIMALITY TESTING



# GENERATING PRIMES

- Preliminarily, set the size of the modulus $n$
  - ***Assume a modulus represented by a bitvector of B bits***
  - Typically, we have *B=1024, 2048, 3072, 4096,...*
- To generate the prime $p$
  1. Using a high-quality pseudo-random number generator, generate a random number of size *B/2* bits
  2. Set the lowest bit of the integer generated above
     - This ensures that the number is odd
  3. Set the two highest bits of the integer
     - This ensures that the highest bits of $n$ are also set so that the number is likely to be large
  4. Check if the resulting integer is prime (e.g., by using the Miller-Rabin test)
  5. If not, increment the integer by 2 and check again
- Generate the prime $q$ in a similar way
  - Check that the two generated integers are distinct

47

# KEY GENERATION

1. ***Generate two distinct primes p and q***
2. Calculate the modulus $n = p * q$
3. Calculate the totient $\phi(n) = (p - 1) * (q - 1)$
4. Select for ***public exponent*** an integer $e$ such that $1<e<\phi(n)$ and $gcd(\phi(n), e) = 1$
5. Calculate for the **private exponent** a value $d$ such that $d = e^{-1} \bmod \phi(n)$
6. Set public key to {e, n}
7. Set private key to {d, n}



# SELECTING THE PUBLIC EXPONENT

- Recall that the **public exponent** should be an integer $e$ such that $1<e<\phi(n)$ and $gcd(\phi(n), e) = 1$
- Recall also that $q$
- So, from $gcd(\phi(n$
  - $gcd(p - 1, e) =$
  - $gcd(q - 1, e) =$
- To simplify comp
  - is prime
  - has as few bits as possible equal to 1 for fast multiplication
  - is cryptographically secure
  - Typically, we have $e = 3, 5, 17, 257, 65537 (= 2^{16} + 1)$
    - *These* have only two bits set for fast modular exponentiation

| | | |
|---|---|---|
| | 1 0 1 1 | this is $11_{10}$ |
| × | 1 0 1 0 | this is $10_{10}$ |
| | 0 0 0 0 | 1011 x 0 |
| | 1 0 1 1 | 1011 x 1, shifted one position to the left |
| | 0 0 0 0 | 1011 x 0, shifted two positions to the left |
| + | 1 0 1 1 | 1011 x 1, shifted three positions to the left |
| | 1 1 0 1 1 1 0 | this is $110_{10}$ |

**Exercise**: try to understand why having few bits set speed multiplications up

# HINT TO EXERCISE

- Each time you have a 1, you need to shift and add
- Each time you have a 0, you only need to shift

| | | |
|---|---|---|
| | 1 0 1 1 | this is $11_{10}$ |
| × | 1 0 1 0 | this is $10_{10}$ |
| | 0 0 0 0 | 1011 x 0 |
| | 1 0 1 1 | 1011 x 1, shifted one position to the left |
| | 0 0 0 0 | 1011 x 0, shifted two positions to the left |
| + | 1 0 1 1 | 1011 x 1, shifted three positions to the left |
| | 1 1 0 1 1 1 0 | this is $110_{10}$ |



51

# DIGRESSION ON MODULAR EXPONENTIATION

# MODULAR EXPONENTIATION

- When ***2^(2^n) +1*** is prime, it is said to be a **Fermat number**
- The only known Fermat primes are the first five Fermat numbers: 3, 5, 17, 257, and 65537

- **Encryption**: the plaintext integer *M* is raised to the power *e* modulo *n* to yield the ciphertext integer *C*
  - The exponentiation operation for encryption can be carried out efficiently by simply choosing an appropriate *e*
  - The only condition on e is such that e and $\phi(n)$ are coprime
  - Typical choices for *e* are *3, 5, 17, 257, 65537* that have only a small number of bits set
- **Decryption:** the ciphertext integer *C* is raised to the power *d* modulo *n* to yield the plaintext integer *M*
  - Modular exponentiation for decryption is a much more difficult task since there is no freedom to choose *d* since the value of *d* is determined completely by *e* and *n*
  - Typically, *d* is of roughly the same size as the modulus *n* and is usually a very large integer

- For n = 2, we have 2^(2^n) +1 = 5 = 1 000 1
- The bit-vector representation of 2^(2^n) +1 contains just two 1s



# AN ALGORITHM FOR MODULAR EXPONENTIATION (1)

- Indeed, exponentiation is repeated multiplication of a number by itself…
- Having few bits sets speed up exponentiation but…
- … the process can be further improved…

- We still need to devise a procedure to compute
$$A^B \bmod n$$
- Notice that even for small values for A and B, the value of $A^B$ can be very large
  - **Example**: 7^11 = 1, 977, 326, 743 (a number with 10 digits)
  - Consider, as would be the case in cryptography, *A* has 256 binary digits (that is 77 decimal digits) and *B* has 65,537 binary digits…
  - … even when *B* has only 2 digits (say, B = 17), when *A* has 77 decimal digits, $A^B$ will have 1304 decimal digits
- The calculation of $A^B$ can be sped up by realizing that if *B* can be expressed as a sum of smaller parts, then the result is a product of smaller exponentiations
- We can do this by expressing *B* in binary format…

# AN ALGORITHM FOR MODULAR EXPONENTIATION (2)

- We can use the following binary representation for the exponent $B$:
$$B = b_{k-1}b_{k-2} \dots b_1 b_0$$
- We can thus write $B$ as follows
$$B = \sum_{b_i \neq 0} 2^i$$
- This, in turn, allows us to write $A^B$ as
$$A^{\sum_{b_i \neq 0} 2^i} = \prod_{b_i \neq 0} A^{2^i}$$
- This form of $A^B$ roughly halves the difficulty of computing $A^B$ because, assuming all the bits of $B$ are set, the largest value of $2^i$ will be about half the largest value of $B$



# AN ALGORITHM FOR MODULAR EXPONENTIATION (3)

- We can achieve further simplification by bringing the rules of modular arithmetic into the multiplications on the right:
$$A^B \ mod \ n = \left(\prod_{b_i \neq 0} A^{2^i}\right) mod \ n$$
- Now, observe that $A^{2^i}$ for increasing values of $i$ gets the following values
$$A^{2^0}, A^{2^1}, A^{2^2}, \dots, A^{2^B}$$
that can be expressed as follows
$$A, A^2_{previous}, A^2_{previous}, \dots, A^2_{previous}$$
i.e. instead of calculating each term from scratch, we can calculate each by squaring the previous value

# AN ALGORITHM FOR MODULAR EXPONENTIATION (4)

- By putting together the previous observations, we can define the following algorithm to compute modular exponentiation
- **Input**: integers $A$ and $B$ and modulus $n$
- **Output**: $A^B \ mod \ n$
- **Method**
  1. result = 1
  2. **while** ($B>0$) **do**
     3. **if** the least significant bit of B is 1 **then**
        result = (result * $A$) mod $n$
     4. shift B to the right by one bit
     5. $A = (A * A) \ mod \ n$
  3. **end**
  6. **return** result

- Even with a straightforward implementation of the method, computing for instance $7^{9633196} \ mod \ 9633196$ almost instantaneously yields 117649
- Trying to compute the same value with the standard operations provided by programming languages can take up to several minutes



57

# END OF DIGRESSION ON MODULAR EXPONENTIATION

# ON BEING CRYPTOGRAPHICALLY SECURE

- Small values for $e$, such as 3, are considered **cryptographically insecure**
- To understand why, consider A sends the same message M to three different receivers using their respective public keys that have the same $e = 3$ but different values of $n$, say $n_1, n_2, n_3$, respectively
- Assume that an attacker can intercept all transmissions so that it can collect the following 3 ciphertexts

$$C_1 = M^e mod\ n_1 \qquad C_2 = M^e mod\ n_2 \qquad C_3 = M^e mod\ n_3$$

  with $e=3$
- Assuming that $n_1, n_2, n_3$ are relatively prime on a pairwise basis, the attacker can use the **Chinese Remainder Theorem** to compute $M^3\ mod\ (n_1 * n_2 * n_3)$
- The attacker then can compute $M$ from $M^3$ which is not a computationally complex task



59

# DIGRESSION ON THE CHINESE REMAINDER THEOREM

# CHINESE REMAINDER THEOREM (1)

- Very old (4th century AD) result
- It states that in modulo *M* arithmetic, if *M* can be expressed as a product of *n* integers that are pairwise coprime, then every integer in the set $\{0, 1, 2, \ldots, M-1\}$ can be reconstructed from residues with respect to those *n* numbers
- **Example**
  - Take *M=10=2*5*
  - Now, observe that *9 mod 2 = 1* and *9 mod 5 = 4*
  - The idea is to represent 9 with the pair (1,4)
  - Why this is a good idea will be apparent below…



# CHINESE REMAINDER THEOREM (2)

- Fix a decomposition of *M* in coprime factors, i.e.

$$M = \prod_{i=1}^{k} m_i \text{ for } \gcd(m_i, m_j) = 1 \text{ with } 1 \leq i \neq j \leq k$$

- Notice that the decomposition may not be unique
  - Example: *M=130=5*26=130=2*5*13*
- Represent any integer in $\{1, \ldots, M-1\}$ as a tuple $(a_1, \ldots, a_k)$ where $a_i = M \bmod m_i$ for $i=1,\ldots,k$ (notice that $a \leq a_i < m_i$)
- CRT states that
  - an integer in $\{1, \ldots, M-1\}$ is uniquely associated to a tuple
  - arithmetic operations modulo *M* can be carried out independently on the components of the tuple
    - $A + B \bmod M$ can be mapped to $((a_1 + b_1) \bmod M, \ldots, (a_k + b_k) \bmod M)$
    - Similarly for subtraction and multiplication….

# CHINESE REMAINDER THEOREM (3)

- The interesting part is that it is **possible to reconstruct the result of an arithmetic operation modulo *M* from the results of the arithmetic operations over the tuples obtained by the decomposition of *M***
- To do this, the key observation is the following: given a number $A$ and the corresponding tuple $(a_1, \dots, a_k)$ we have that

$$A = (\sum_{i=1}^{k} a_i * c_i) \; mod \; M$$

where $c_i$ is obtained by multiplying $M_i$ by its multiplicative inverse w.r.t. $m_i$

- We clarify the situation with an example…



# CHINESE REMAINDER THEOREM (4)

**Example**

- Let *M=8,633=89*97*
- Observe that
  - *97 * 78 ≡ 1 (mod 89)*
  - *89 * 12 ≡ 1 (mod 97)*
  - I.e. 78 is the multiplicative inverse of 97 with respect to 89 and 12 is the multiplicative inverse of 89 modulo 97
- We want to compute *2345 + 6789 mod 8633*
  - 2345 → (31, 17) since *2345 mod 89 = 31* and *2345 mod 97 = 17*
  - 6789 → (25, 96) since *6789 mod 89 = 25* and *6789 mod 97 = 96*
  - *(31,17)+(25,96)= (56,16)*
  - *56*97*78+16*89*12 mod 8,633 = 501*

64

# END OF DIGRESSION ON THE CHINESE REMAINDER THEOREM



# KEY GENERATION

1. ***Generate two distinct primes p and q***
2. Calculate the modulus $n = p * q$
3. Calculate the totient $\phi(n) = (p - 1) * (q - 1)$
4. Select for ***public exponent*** an integer $e$ such that $1<e<\phi(n)$ and $gcd(\phi(n), e) = 1$
5. Calculate for the ***private exponent*** a value $d$ such that $d = e^{-1} \bmod \phi(n)$
6. Set public key to {e, n}
7. Set private key to {d, n}

# SELECTING THE PRIVATE EXPONENT

- Recall that we need to find a value *d* of the private exponent such that
$$e * d \equiv 1 \ (mod \ \phi(n))$$
- This means to compute the **modular inversion** of the public exponent *e*, i.e.
$$d = e^{-1} \ mod \ \phi(n)$$
- Since *d* is the multiplicative inverse of *e* modulo $\phi(n)$, we can use the **Extended Euclid's Algorithm** for calculating *d*
  - Recall that $\phi(n) = (p-1) * (q-1)$
- **The main source of security in RSA is keeping both *p* and *q* secret** and thus also keeping $\phi(n)$ secret
  - Indeed, knowing either will reveal the other, i.e if the factors *p* and *q* are known, then also $\phi(n)$ is known (see above) and then also *n* can be determined since both its factors *p* and *q* can be identified



67

# DIGRESSION ON THE EXTENDED EUCLID'S ALGORITHM

# EUCLID'S ALGORITHM FOR GCD

- Very old algorithm
- It is based on the following properties
  - $\gcd(a, a) = a$
  - If *b* divides *a*, then $\gcd(a, b) = b$
  - $\gcd(a, 0) = a$ (this is so because it is always the case that *a* divides 0)
    - This is the base case of the recursion
- The recursive step of the algorithm is the following
  $$\gcd(a, b) = \gcd(b, a \bmod b)$$
- **Example**
  - $\gcd(70,38) = \gcd(38, 70 \bmod 38) = \gcd(38,32) = \gcd(32, 38 \bmod 32) = \gcd(32, 6) = \gcd(6, 32 \bmod 6) = \gcd(6,2) = \gcd(2, 6 \bmod 2) = \gcd(2,0) = 2$



# FINDING MULTIPLICATIVE INVERSES (1)

- The multiplicative inverse of *a* with respect to *n* (with *a*<*n*) is the integer *b*<*n* such that $a * b \equiv 1 \pmod n$
- The multiplicative inverse exists for each integer *a*<*n* such that *a* and *n* are coprime, i.e. $\gcd(a, n) = 1$
  - Notice that if *n* is a prime, then it is always possible to find a multiplicative inverse for any non-zero integer *a*<*n*
- **Bezout's identity**

  $\gcd(a, b) = x * a + y * b$ for some integers *x* and *y* (positive or negative)
  - Notice that *x* and *y* need not be unique
- **Example**
  - Take *a = 16* and *b = 6*
  - *gcd(16, 6) = 2* = (−1) * 16 + 3 * 6 = 2 * 16 + (−5) * 6

# FINDING MULTIPLICATIVE INVERSES (2)

- It is possible to use Bezout's identity to find multiplicative inverses
- Recall that *b* is the multiplicative inverse of *a* w.r.t. *n* if $\gcd(a, n) = 1$
- By Bezout's identity, we derive that there exist *x* and *y* such that
$$x * a + y * n = 1$$
- If we take the equality above modulo *n*, we have
$$x * a \bmod n + y * n \bmod n = 1 \bmod n$$
that simplifies to $x * a \bmod n = 1$

  i.e. the value of *x* should be the multiplicative inverse of *a* with respect to *n*
- This suggests an idea to extend Euclid's algorithm to compute multiplicate inverses…



# FINDING MULTIPLICATIVE INVERSES (3)

- Use the Euclid's algorithm with the following modifications
  - at each step, write the expression of the remainder in the form a * x + n * y
  - when the remainder becomes 1 (which will happen only when *a* and *n* are relatively prime), *x* will automatically be the multiplicative inverse of *a* w.r.t. *n*
- **Example**
  - Find the multiplicative inverse of 32 with respect to 17
  - *gcd(32,17)* =
    - = *gcd(17,15)* | residue 15 = 1*32 – 1*17
    - = gcd(15, 2) | residue 2 = 1*17 – 1*15 = 1*17 – 1*(1*32 – 1*17) = 2*17-1*32
    - = gcd(2,1) | residue 1 = 1*15 – 7*2 = 1*(1*32 – 1*17) – 7*( (-1)*32 + 2*17 ) = 8*32 – 15*17
  - We can conclude that the multiplicative inverse of 32 with respect to 17 is 8
    - Indeed, it is easy to check that *32*8 mod 17 = 1*

72

# END OF DIGRESSION ON THE EXTENDED EUCLID'S ALGORITHM



73

# SOME REMARKS ON THE SECURITY OF RSA

**Padding** is the process of preparing a message for encryption (or signing) by adding random material at the beginning, middle or end of a message

# WEAKNESS OF THE BASIC ALGORITHM

- Basic RSA algorithm = encryption/decryption scheme implemented as described above
- Basic RSA algorithm would be much too vulnerable to several kinds of attacks
- **Simple vulnerability**
- *Example*
  - Public key uses exponent 3
  - Entity A sends only very short messages to other entities
  - If a message $M$ is short enough, the ciphertext integer $C = M^3 \bmod n$ will be smaller than the modulus $n$
  - Then, an attacker will be able to recover the plaintext integer $M$ simply by taking the cube-root of $C$ by using an appropriate algorithm for extracting the $n$-th root of an integer
  - To mitigate this attack, plaintext integers should be padded so to span the full length of the modulus
  - With **padding**, when the message $M$ is raised to the power of the public exponent (even a small public exponent like 3), the result would exceed the modulus $n$ and $C$ would be the remainder modulo $n$
  - Since an algorithm for the extraction of the $n$-th root does not exist for modular arithmetic, attackers would not be able to recover $M$ even if it is short



# CHOSEN CIPHERTEXT ATTACKS (1)

- Consider entity A sends a ciphertext message $C = M^e \bmod n$ to B with
  - $\{e,n\}$ being the public key of B
  - {d,n} being the private key of B
- **Moore's attack**
- If the attacker Eve intercepts $C$, it can guess the plaintext $M$ without knowing the private key of B as follows
  - Eve randomly chooses an integer $s$ with a multiplicative inverse $s^{-1}$ w.r.t. the modulus $n$
  - Eve constructs a new message $C' = s^e * C \bmod n$ and sends it to B
  - B decrypts the message $C'$
  - B sends back the signed ciphertext $M' = C'^d \bmod n = (s^e * C \bmod n)^d \bmod n = s^{e*d} * C^d \bmod n = s * M \bmod n$ (because the attacker has somehow convinced B to do so)
  - At this point, Eve will be able to recover the original message $M$ by multiplying $M'$ by $s^{-1}$:

$$M' * s^{-1} = (s * M \bmod n) * s^{-1} = M$$

# CHOSEN CIPHERTEXT ATTACKS (2)

- Preliminarly, let us first consider **PKCS#1v1.5**
  - PKCS = Public Key Cryptography Standards
  - PKCS are a group of standards initially promulgated by RSA
  - Some of the PKCS have now entered an official standardization process by being included in RFCs
    - For instance PKCS#1 is now included in RFC8017 (https://tools.ietf.org/html/rfc8017) containing "*recommendations for the implementation of public-key cryptography based on the RSA algorithm, covering cryptographic primitives, encryption schemes, signature schemes […] syntax for representing keys and for identifying the schemes*"
  - Version 1.5 of PKCS#1 is one of the most widely used of the PKCS standards, still today, despite being published in 1993



# ABOUT PKCS#1V1.5

- Encryption is applied to a block of bytes, called **Encryption Blocks** (**EB**s) whose structure is the following

  00 || BT || PS || 00 || D

  where
  - || denotes concatenation
  - the numeric '00' stands for a byte whose value is 0
  - BT means a one-byte integer that designates the type of EB
    - The value of 'BT' is the integer 2 for encryption with RSA
    - The values 0 and 1 for 'BT' are for generating digital signatures
  - PS means a pseudo-randomly generated "Padding String"
  - D stands for the data bytes

- Additionally, **PS shall contain at least 8 bytes** for security reasons
- Thus, **the minimum value for k is 12 bytes** because
  - 2 bytes for two 00
  - 1 byte for BT
  - 8 for PS
  - 1 leftover for D

- **The size of the EB is k bytes and is equal to the size of the modulus in bytes**

# CHOSEN CIPHERTEXT ATTACKS (3)

- **Bleichenbacher attack (1998)**
- The attacker Eve has access to an oracle that tells if any ciphertext is conformant to the PKCS standard
  - The oracle is easy to implement
    - Upon reception of the ciphertext integer, this is converted to a block of *k* bytes
    - The first byte is checked to be 00
    - The second byte (BT) is checked to be the integer 2
    - … and so on
- Eve craft a sequence of ciphertexts to be sent to the oracle
- It is possible to show that Eve gains information about the ciphertext by making the sequence of ciphertexts adapting (i.e. the next ciphertext is crafted according to the previous answers of the oracle)



# CHOSEN CIPHERTEXT ATTACKS (3)

- **Bleichenbacher attack (continued)**
- Eve crafts a sequence of randomly selected integers *s* to form a candidate sequence of ciphertexts $C' = s^e * C \ mod \ n$
- Eve chooses the integers *s* one at a time, forms the ciphertext $C'$ and sends it to the oracle to find out if $C'$ is conformant to the PKCS1 standard or not, i.e. in particular if its first two bytes have the integer values of 0 and 2
- Each positive answer from the oracle allows the attacker to enlarge the size of *s* and make an increasingly narrower estimate for the value of the plaintext integer *M* that corresponds to the original *C*
- The sequence of values for *s* ends when the estimated value for *M* is just one number

Variants of this attack are still relevant for today version of TLS (1.3)
https://www.zdnet.com/article/new-tls-encryption-busting-attack-also-impacts-the-newer-tls-1-3/

# MORE DETAILS ON BLEICHENBACHER (1)

- Alice sends an encrypted session key to Bob using the PKCS#1v1.5 standard
  - This implies that the message M that Alice is going to send Bob has the following format

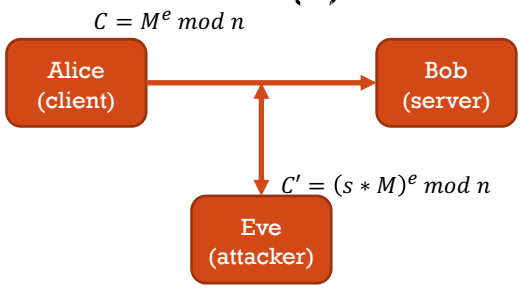
    00 || 02 || PS || 00 || Kab

    where PS is some padding and Kab is the session key
- Eve intercepts the encrypted message M containing the session key Kab, i.e.

  $$C = M^e \bmod n$$
- Eve then sends the following message

  $$C' = s^e * M^e \bmod n = (s * M)^e \bmod n$$

  to the server …



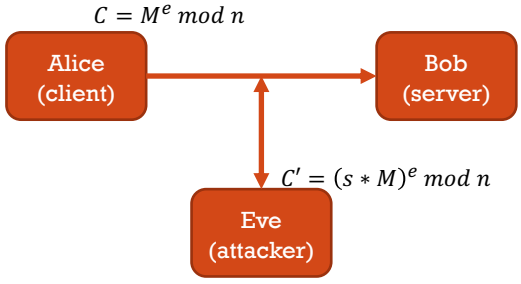

- {e,n} is the public key of Alice
- s is an integer selected by Eve



# MORE DETAILS ON BLEICHENBACHER (2)

- Bob decrypts *C'* to get

  $$M' = C'^d \bmod n = (s * M)^{e*d} \bmod n = (s * M) \bmod n$$
- Bob checks if *M'* complies with the PKCS#1v1.5 and in particular checks if the first two bytes are 00 and 02
  - Bob does this invoking a simple function which can be thought of as an oracle
- When *M'* complies with the standard, then to get *M* and hence also the shared key *Kab*, it is enough to divide *M'* by *s* (modulo *n*)
- Indeed, Eve should be quite lucky to get the right value of s that yields a message *M'* that is compliant with the standard PKCS#1v1.5…
- … or Eve can repeatedly select a different value of *s* until it gets a positive answer from the oracle checking compliance with the standard




- {e,n} is the public key of Alice
- s is an integer selected by Eve

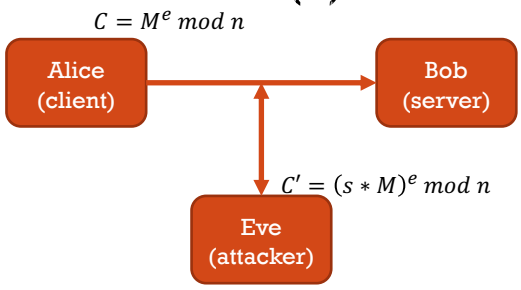
# MORE DETAILS ON BLEICHENBACHER (3)

- Sooner or later, Eve will select a value of *s* such that *(s*M mod n)* is compliant with the PKCS#1v1.5 standard and in particular it starts with 00 and 02 when considering the integer as a sequence of bytes
- Now, this means that *(s*M mod n)* is such that

$$2 * B \leq s * M - r * n \leq 3 * B$$

for some value *r* where $B = 2^{8*(l-2)}$ with *l* being the size of the modulus *n* in bytes

  - To understand why the above inequalities hold, recall the definition of the modulus operation and the format of the PKCS#1v1.5 where the first two bytes are fixed values 00 and 02
- Since both M and r are unknown, we need to work a bit more to establish their values: this is done by finding other values of *s* that generates messages that are compliant with the PKCS#1v1.5 standard…




- {e,n} is the public key of Alice
- s is an integer selected by Eve

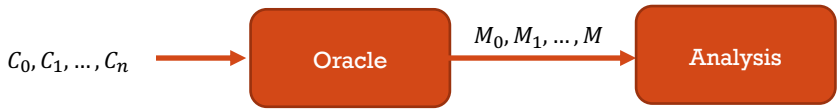


… the details for doing this can be found in http://archiv.infsec.ethz.ch/education/fs08/secsem/bleichenbacher98.pdf

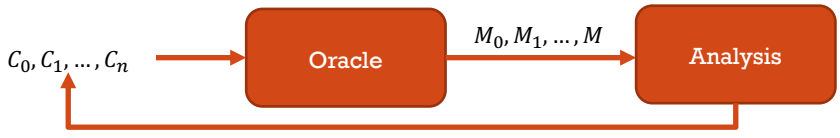


# TYPES OF CHOSEN CIPHERTEXT ATTACKS (1)

CCA1




CCA2




The **padding oracles**, as in Bleichenbacher's attack, take ciphertexts as input and reveal whether or not the padding is correct after their decryption

- An **oracle** is a black-box that will take an input and answer a specific, fixed question regarding its input, without being restricted in terms of knowledge
- It can be used by any user indifferently, including any attacker

# TYPES OF CHOSEN CIPHERTEXT ATTACKS (2)

- Two different types of chosen ciphertext attacks: CCA1 and CCA2
- CCA1 (also called **passive chosen ciphertext attack)**
  - The attacker can consult the decryption oracle an arbitrary number of times, but only until the attacker has acquired the ciphertext *C* through eavesdropping
- CCA2 (also called **adaptive chosen ciphertext attack**)
  - The attacker can continue to consult the oracle even after seeing the ciphertext *C*
- Bleichenbacher attack is a CCA2
  - This implies that **PKCS#1v1.5 is not CCA2 secure**
- RSA is made resistant to CCA2 when the padding bytes are set according to OAEP (Optimal Asymmetric Encryption Padding)
- Unlike for the PKCS#1v1.5 standard, EBs have no structure…



# LOW ENTROPY IN KEYS (1)

- The entropy of a random number generator is at its highest if all numbers are equally likely to be produced within the range of numbers that the output is designed for
- **Example**
  - if a generator can produce 512-bit random numbers with equal probability, its entropy is at its maximum and it equals 512 bits
  - If the probabilities associated with the output random numbers are nonuniform, the entropy will be less than 512 bits
- The greater the nonuniformity of the probability distribution, the smaller the entropy
- The entropy is zero for deterministic output

# LOW ENTROPY IN KEYS (2)

- Consider the situation in which an attacker has identified a common factor $p$ of two moduli $n_1, n_2$ by using Euclid algorithm (with quadratic complexity in the number of bits representing the integers)
  - This implies that $p$ is the greatest common divisor of $n_1, n_2$
- We can easily derive factors $q_1, q_2$ by division, i.e. $n_1 = p * q_1$ and $n_2 = p * q_2$
- It is then immediate to compute the private keys by $d_i = (e_i - 1)\ mod\ (p-1) * (q_i - 1)$ for $i=1,2$ (notice that $e_i$ is the public exponent and it is readily available)
- **How likely it is for an attacker to find a pair of RSA moduli with a common factor?**
- Unfortunately, the answer is: **quite likely!**
- This is so because of the availability of port scanners that can carry out a full SSL/TLS and SSH handshake and fetch the certificates used by the TLS/SSL hosts and the host keys used by the SSH servers very quickly and on a large scale…



# LOW ENTROPY IN KEYS (3)

- In a Usenix 2012 paper (available at https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/heninger), it was experimentally shown how to harvest over **5 million TLS/SSL certificates** and around **4 million RSA-based SSH host keys** with scanning for two days the Internet ports
- By using appropriate algorithms, the authors were able to find the factors that any of the moduli shared with any of the other moduli from the collected keys
- They were able to compute the private keys for
  - 0.50% of the TLS/SSL servers (around 25,000)
  - 0.03% of the SSH servers (around 1,200)
- This implies that random number generators used in key generation should have high enough entropy so that **every modulus is unique vis-a-vis the moduli used by any other communication device** anywhere on earth

# LOW ENTROPY IN KEYS (4)

- While the prescription above is followed by most computers that we currently use, this is not the case for a large number of (so called) **headless communication devices** such as routers, firewalls, server management cards, etc
- A very large number of these devices use inadequate software sources of entropy for the random bytes needed to generate prime numbers
- The problem with such software sources of entropy is at boot time when they are initialized
  - as they are not very well equipped to supply high-entropy random bytes
  - at the same time, the network interface needs to create keys



# MATHEMATICAL ATTACKS (1)

- The security of RSA depends critically on the fact that factoring the modulus into its prime factors can be extremely difficult whereas it is easy to multiply two **large** primes to construct the modulus
  - Functions that are easy to compute in one direction but that cannot be easily inverted without special information are known as **trapdoor** functions
- Trying to break RSA by developing an integer factorization solution for the moduli involved is known as a mathematical attack
  - This amounts to figuring out the prime factors $p$ and $q$ of the modulus $n$
  - As already observed, knowing $p$ and $q$ immediately yields the private exponent $d = (e-1)\ mod\ (p-1) * (q-1)$ for decryption
- A **semiprime** is a number obtained by the product of two prime numbers
  - So, any modulus should be a semiprime
  - The largest known semiprime is $(2^{230,402,457}-1)^2$ [this number has 18 millions digits]

# MATHEMATICAL ATTACKS (2)

- Various mathematical techniques have been developed for solving the integer factorization problem involving large numbers
- We just mention some of the most important methods without the goal of providing a detailed description
  - **Trial division**
    - It works by dividing the number to be factorized by successively larger integers
    - It works for integers with up to 12 decimal digits
  - **Fermat's Factorization Method**
    - Based on the observation that every odd number $n$ that has two non-trivial factors can be expressed as a difference of two squares, i.e. $n = x^2 - y^2 = (x + y) * (x - y)$
      - I.e. the two factors are $x + y$ and $x - y$
    - It works fast if $n$ has a factor close to its square-root



# MATHEMATICAL ATTACKS (3)

- **Sieve based methods**
  - The oldest known sieve is the sieve of Eratosthenes for generating prime numbers
  - To find all the prime integers up to a number
    - write down the numbers successively (starting with the number 2) in an array-like display
    - Cross out all the numbers divisible by 2 and add 2 to the list of primes
    - Cross out all the numbers divisible by 3 and add 3 to the list of primes
    - …
  - Modern sieves for fast factorization are quadratic sieve, number field sieve, etc
  - For instance, the quadratic sieve method is the fastest for integers under 110 decimal digits
- **Pollard method**
  - **…**

# ON FACTORIZING LARGE NUMBERS

- In the past, RSA sponsored a factoring challenge
  - Started in 1991
  - Ended in 2007
- Many of the large numbers put forward by RSA for factoring have still not been factored and are not expected to be factored any time soon
- Challenges are denoted by RSA-NNNN where NNNN is the number of bits needed for the binary representation of the integer to be factored
- Details available at https://en.wikipedia.org/wiki/RSA_Factoring_Challenge



93

# PRAGMATICS ON KEYS

# GENERAL REMARKS ON KEYS (1)

- The representation depends on the protocol
  - SSH and TLS use very different representation
- The size of the key in the RSA algorithm typically refers to the size of the modulus integer in bits even though keys are pairs of the form $\{e,n\}$ and $\{d,n\}$ so including also the public and private exponents
  - For instance, when talking about "a RSA implementation that provides 1024 bits of security," this means that we are discussing a RSA algorithm that uses modulus of size1024 bits
  - Observe that 1024 bits can be stored in only 128 bytes of memory and yet the decimal value of the integer represented by these 128 bytes can be very very very large
- RSA recommends that the two primes that compose the modulus should be roughly of equal length
  - So if one wants to use 1024-bit RSA encryption, then the modulus integer will have a 1024 bit presentation; this, in turn, implies that the two primes must be roughly 512 bits each



# GENERAL REMARKS ON KEYS (2)

- **Doubling the size of the key**, i.e. the size of the modulus, will, in general, **increase** the **time** required
  - for encryption **by a factor of 4** and
  - for decryption **by a factor of 8**
- Operations involving the private key are less affected because the public key exponent $e$ does not have to change as the key size increases while the private key exponent $d$ changes in direct proportion to the size of the modulus
- **Key generation time increases by a factor of 16** when the size of the key (meaning the size of the modulus) is doubled
  - Notice that key generation is relatively infrequent

# REPRESENTING KEYS (1)

Base64 encoding is a group of binary-to-text encoding schemes that represent binary data in an ASCII string format

- **For the public key**, in addition to storing the encryption exponent and the modulus, the key may also include information such as
  - the time period of validity
  - the name of the algorithm used for key generation
  - etc
- **For the private key**, in addition to storing the decryption exponent and the modulus, the key may include additional information along the same lines as for the public key, and, additionally, the corresponding public key also
- Typically, the formats require the keys to be stored using **Base64 encoding** so that they can be displayed using printable characters



# REPRESENTING KEYS (2)

For SSH, the standard for key representation is in RFC4253, available at https://tools.ietf.org/html/rfc4253

- **ssh-keygen** is a standard component of the Secure Shell (SSH) protocol suite found on Unix, Unix-like and Microsoft Windows computer systems used to establish secure shell sessions between remote computers over insecure networks, through the use of various cryptographic techniques
  - The ssh-keygen utility is used to generate, manage, and convert authentication keys
- **Example**
  - **ssh-keygen -t rsa**
  - A passphrase is required to generate the pairs of public and private keys
  - The generated keys allow a laptop to establish SSH connections with machines elsewhere from virtually anywhere in the world without the user having to log in explicitly with a password if…
  - … one copies the public key that is generated into the file storing the authorized keys in the user own account of the remote machine to which the user wants SSH access

# REPRESENTING KEYS (3)

Chunk of Base64 encoded data containing the public exponent and the modulus

Key type

- **Example of a public key**
- ssh-rsa
  AAAAB3NzaC1yc2EAAAADAQABAAABgQDrj3O8QFLG+nFRudxzt7ruA6h4ec0WT
  UhfhfnZZ8kcdZErsIFq4BJ++TWodrWIWsdXqGccm0AWOpPkDKUt+wCf9ptRsZtYOV
  imPkf3lnG+qANzQWuj6esRkaSDq3xxuenJFHHq68XrsY1xpEM8twm9TsbgHwm7FU
  VJv5Z4tkSyqiJm8WwoopBMJNtWrOYxHq5Cbr86syPG/jbf5UYNyvsfZcrngahy5cLBE
  dfHz64qABfYbNdhAWLoSH3p4KfObpFGqLuXuDO5BZsIGFuQqWsPngq+lPl0t2ywY
  6wQNZ0JPZKP/k1V4RlTSO4SqfDMfzuz/1LArK38AIoy3OjSLZrCgBHEu4M9DO95NUc
  kvfUew3kK0ykEP3bA07loFfO7252TZAQ2TiAq5aT8oGkyjPOk4oE607jlia8yM8PNMf
  REY8zRa5HcuUxL5u8Mc/s1yFCuwRCWvPgnukOFVWFIbqCM0yGCLqJUDQ5sAraR
  +ibwkfVib3AV9J3aOfPvGo3Uffk= Silvio@Silvios-MacBook-Air.local

Comment

- Public exponent: 35
- Modulus: 2899223926596568013…9357691872217



99

# RSA: CONCLUDING REMARKS

# FINAL REMARKS (1)

- Assuming to use
  - the best possible random number generator to create candidates for the primes that are needed to compute the modulus and
  - a recent version of the RSA scheme that is resistant to the chosen ciphertext attacks

  the security of RSA depends critically on the difficulty of factoring large integers
- As integer factorization algorithms have become more and more powerful over the years, RSA must rely on increasingly larger values for the integer modulus and, therefore, increasingly longer encryption keys
- Nowadays, it is unlikely to use a key whose length is shorter than 1024 bits
  - Someone recommends 2048 or even 4096 bit keys



# FINAL REMARKS (2)

- Comparison between RSA and other cryptographic primitives

| Symmetric key algorithm | Key size for symmetric key cryptography | Comparable key size for RSA algorithm |
|---|---|---|
| 2-Key 3DES | 112 | 1,024 |
| 3-Key 3DES | 168 | 2,048 |
| AES-128 | 128 | 3,062 |
| AES-192 | 192 | 7,680 |
| AES-256 | 256 | 15,360 |

More details available at
https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf

# FINAL REMARKS (3)

- The table in the previous slide tells us that the computational overhead of RSA encryption/decryption goes up as the size of the modulus increases
- This makes RSA inappropriate for encryption/decryption of actual message content for high data-rate communication channel
- RSA plays a key role for the exchange of secret keys that can subsequently be used for the more traditional (and much faster) symmetric-key encryption and decryption of the message content

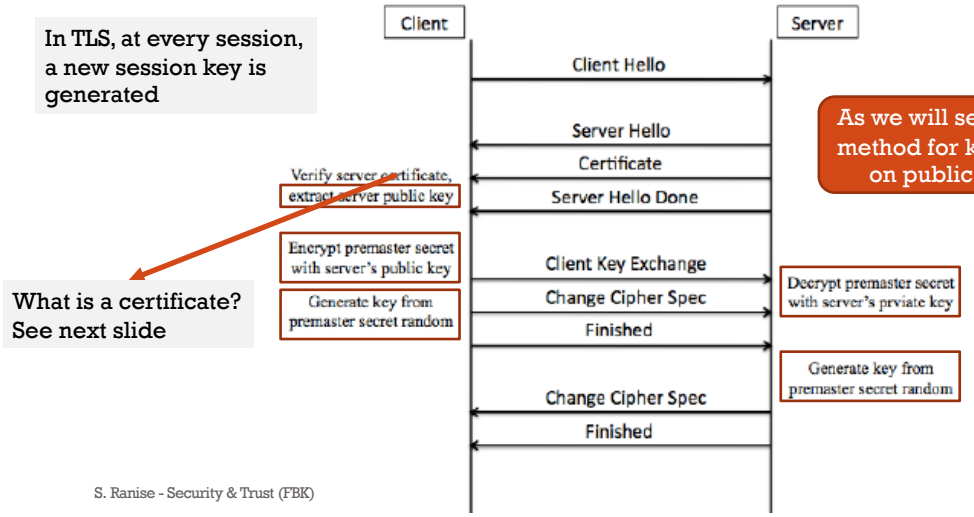


103

# KEY EXCHANGE WITH PUBLIC KEY CRYPTOGRAPHY

# EXAMPLE: RSA KEY EXCHANGE IN TLS

In TLS, at every session, a new session key is generated




As we will see, RSA is not the only method for key exchanged based on public key cryptography

What is a certificate? See next slide

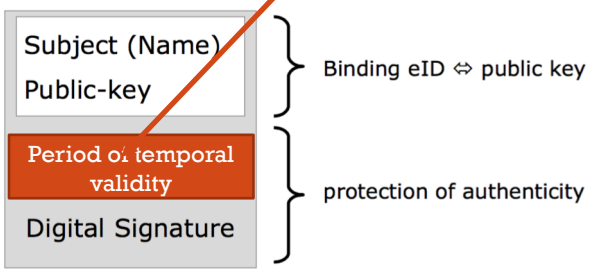


# CERTIFICATES FOR AUTHENTICATING PUBLIC KEYS

- Bind public key to electronic identity
- Seal the binding

- **Public key certificates are data structures that bind public key values to subjects**
- The binding is asserted by having a trusted CA digitally sign each certificate ... [RFC 5280]
  https://tools.ietf.org/html/rfc5280

- Reduced from 5 to 3 and then to 2 years…
- Plan to cap the validity period of certificates for browsing to 1 year since
  - Any security updates would be rolled out more quickly
  - Higher security of servers as keys are generated more frequently and thus their leakage would have less impact
- Reduction to 1 year suspended because of the pandemics




The standard for certificates is X.509
See, e.g., https://en.wikipedia.org/wiki/X.509 for more details

About the importance of forward secrecy for privacy protection (also *vis à vis* governmental agencies): https://www.eff.org/deeplinks/2013/08/pushing-perfect-forward-secrecy-important-web-privacy-protection

# RSA AND LACK OF FORWARD SECRECY (1)

- **(Perfect) Forward Secrecy is a property of key-exchange protocols in which the exposure of long-term keying material, used in the protocol to negotiate session keys, does not compromise the secrecy of session keys established before the exposure**
- The key sharing protocol based on RSA does not provide forward secrecy
- To see why, consider the situation in which an attacker records all encrypted messages between a client and a server
- In this way, the attacker also records the messages that the client and the server exchange during the key exchange protocol:
  - a client gets the server's certificate to …
  - … authenticate the server and to…
  - …extract the server's RSA public key for creating a secret session key
  - The client generates and then sends a shared key encrypted with the server's public key

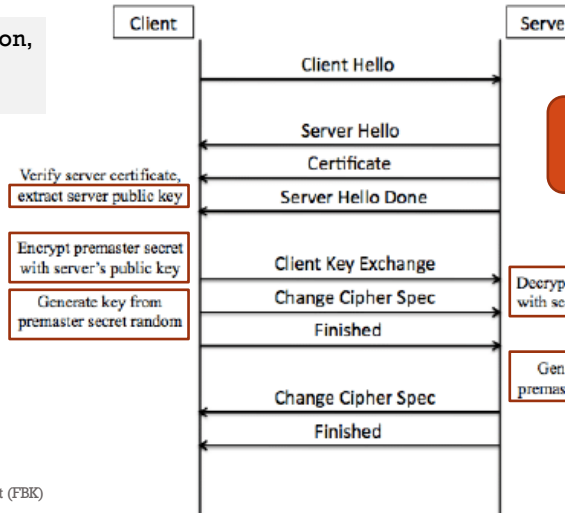


# RSA AND LACK OF FORWARD SECRECY (2)

- If the attacker can get its hands on the server's private key…
- … it will immediately be able to decrypt all the messages that the client and the server has exchanged during a session because…
- … it is able to decrypt the message, encrypted with the server's public key, containing the session key that has been generated by the client
  - Notice that the attacker can do this for any session whose messages it has recorded!
- To understand the likelihood of this attack, we need to understand how likely it is that a private is leaked…
- … unfortunately, it is likely and it has happened in the past because of
  - Insider attackers (e.g., disgruntled employees exfiltrating the private key of company servers)
  - TLS heartbleed attack tthat was able to dump arbitrary memory portion of the server possibly including sensitive information such as its private key

# AGAIN: RSA KEY EXCHANGE IN TLS

In TLS, at every session, a new session key is generated




As we will see, RSA is not the only method for key exchanged based on public key cryptography

If the server's private key is leaked, then also all past session keys can be retrieved if messages are recorded…



# RSA AND LACK OF FORWARD SECRECY (3)

- We conclude that the basic RSA algorithm makes it possible to carry out the exploit described above because it lacks forward secrecy
- Whether or not this vulnerability in a given server-client interaction is a serious matter depends on the nature and the lifetime of the data exchanged between the two endpoints
  - Intuitively, if the data exchanged are quite sensitive but such data is useful for a short time (i.e. they have a limited lifetime), then an attacker will be likely to decrypt the data only after the period in which they are useful
- To avoid this problem, we need to come up with a method to share a session key that avoids to send the key over the channel!
- **This is the magic of Diffie and Hellman…**
  - As we will see, the key exchange schema of Diffie and Hellman is vulnerable to a Man-In-The-Middle attack that can be mitigated by using RSA for the authentication of end-points

110

# THE DIFFIE-HELLMAN KEY EXCHANGE ALGORITHM



## DIFFIE-HELLMAN: IDEA

- Given the following two public parameters (i.e. shared between the parties willing to agree on a shared session key):
  - $p$ is a prime
  - $g<p$ is a group generator (what this means is explained in the digression below)
- The Diffie-Hellman algorithm for key exchange between two parties A and B is
  1. A→B : $g^a \ mod \ p$
  2. B→A : $g^b \ mod \ p$
- A can compute the (pre-)shared key with B as $\left(g^b \ mod \ p\right)^a$
- B can compute the (pre-)shared key with A as $(g^a \ mod \ p)^b$
- It turns out that $\left(g^b \ mod \ p\right)^a = g^{b*a} \ mod \ p = g^{a*b} \ mod \ p = (g^a \ mod \ p)^b$

- Notice that the (pre-)shared key has been never sent over the channel!
- This is crucial for forward secrecy



111

112

# DIGRESSION ON DISCRETE LOGARITHM



# INTRODUCTION

- The **security of RSA** is based on the observation that while it is computationally efficient to perform multiplications of even large integers, it is infeasible or computationally very expensive to **compute the prime factors of large integers**
  - Notice that it is not known if there is an alternative way to attack RSA
  - In other words, the security of RSA is not equivalent to the problem of factoring large integers
- The **security of Diffie-Hellman** is based on the observation that while it is computationally efficient to calculate powers of even large integers, it is infeasible or computationally very expensive to **compute the discrete logarithm of large integers**
- In order to define the notion of discrete logarithm, we need to introduce some notions…

# BASIC NOTIONS (1)

A **group** is a set equipped with a binary operation that combines any two elements to form a third element in such a way that four conditions are satisfied:
closure, associativity, identity and invertibility

- For any positive integer *N*, the set of all integers $i < N$ that are coprime to *N* form a group with modulo *N* multiplication as the group operator
- **Example**: *N=8*
  - The set {1, 3, 5, 7} forms a group with modulo 8 multiplication
  - Closure: for instance, 3 * 5 mod 8 = 7, 3 * 7 mod 8 = 5, 5 * 7 mod 8 = 3
  - 1 is the identity: 1 * 3 mod 8 = 3, 1 * 5 mod 8 = 5, 1 * 7 mod 8 = 7
  - Every element has an inverse with respect to the identity: for instance, 3 * 3 mod 8 = 1, 5 * 5 mod 8 = 1, 7 * 7 mod 8 = 1
  - Associativity is not difficult to check
- When *N=p* with *p* a prime, the group above is denoted with $Z_p^*$ and consists of all the integers in the set {*1, 2, …, p-1*} that are coprime with *p* since this is prime
  - $Z_p^*$ is called the multiplicative group of order *p-1* (where the order is the number of elements of the group, i.e. the cardinality of the set of support)



# BASIC NOTIONS (2)

- For some *N*, the set of support of the group with modulo *N* multiplication contains an element whose various powers (computed modulo *N*) are all distinct and span the entire set of support
  - Such an element is called the **primitive element**, the **primitive root modulo *N***, or the **generator** of the group with modulo *N* multiplication
- **Example**: *N=9*
  - The set of support of the group with modulo 9 multiplication is {1, 2, 4, 5, 7, 8}
  - It is possible to show that 2 is the primitive element or generator
    - *2^0 mod 9 = 1, 2^1 mod 9 = 2, 2^2 mod 9 = 4, 2^3 mod 9 = 8*
    - *2^4 mod 9 = 7, 2^5 mod 9 = 5, 2^6 mod 9 = 1, 2^7 mod 9 = 4*
    - *…*
- Primitive roots modulo *N* can be used to define discrete logarithm…

# DISCRETE LOGARITHM

- Recall that $x^y = z$ and $\log_x z = y$
- Similarly, we can define $x^y \equiv z \ (mod\ N)$ and $dlog_{x,N}\ z = y$
- Recall again the example above:
  - *2^0 mod 9 = 1, 2^1 mod 9 = 2, 2^2 mod 9 = 4, 2^3 mod 9 = 8*
  - *2^4 mod 9 = 7, 2^5 mod 9 = 5, 2^6 mod 9 = 1, 2^7 mod 9 = 4*
  - We can then define
    - $dlog_{2,9}\ 1 = 0, dlog_{2,9}\ 2 = 1, dlog_{2,9}\ 4 = 2, dlog_{2,9}\ 8 = 3$
    - $dlog_{2,9}\ 7 = 4, dlog_{2,9}\ 5 = 5, dlog_{2,9}\ 1 = 6, dlog_{2,9}\ 4 = 7$
- Observe that **the unique discrete logarithm mod *N* to some base *a* exists only if *a* is a primitive root modulo *N***



# REMARKS ON $Z_p^*$ (1)

- It is a cyclic group, i.e. each element can be expressed as
  $$g^i \ mod\ p$$
  for some *i* and *g*
- **Example**: $Z_{17}^*$ is a cyclic group for *g=3*
  - This means that if we compute $3^i \ mod\ 17$ for *i=0,1,…* it is possible to get all possible elements in the support set of $Z_{17}^*$, i.e. $\{1, 2, …., 16\}$
- A subset of the support set of $Z_p^*$ is a **cyclic subgroup** if the group operator is multiplication modulo *p* and all the elements of the subgroup can be generated through the powers of one of the elements of the subgroup
- **Example**: $\{1, 2, 4, 8, 16, 15, 13, 9\}$ is a cyclic subgroup for *g=2*
  - It is easy to check this by computing $2^i \ mod\ 17$ for *i=0,1,…*

# REMARKS ON $Z_p^*$ (2)

- **Lagrange's theorem in Group Theory**
  - If *M* is the order of a cyclic subgroup of $Z_p^*$, then *M* will be a divisor of $p - 1$
- **Example**
  - In the previous example, we have that {1, 2, 4, 8, 16, 15, 13, 9} is a cyclic subgroup for *g=2*
  - The order of the subgroup is the cardinality of the set of support, i.e. *M=8*
  - Indeed, *M=8* is a divisor of *p-1=17-1=16*
- **Property of cyclic group of order *M***
  - $g^M \equiv 1 \ (mod \ p)$
- **Example**
  - From the example above: $2^8 \equiv 1 \ (mod \ 17)$ since *2^8 mod 17 = 256 mod 17 = 1*



119

# END OF DIGRESSION ON DISCRETE LOGARITHM

120

# DIFFIE-HELLMAN: DETAILS



# FROM $Z_p^*$ TO DIFFIE-HELLMAN (1)

- We are interested in those cyclic subgroups of $Z_p^*$ whose order $M$ is large
- More precisely, we are interested in picking a generator $g$ such that the order $M$ of the induced cyclic subgroup is a large prime that is also a factor of $p - 1$
- **The security of the Diffie-Hellman algorithm crucially depends on the size of the cyclic subgroup**
- The following three parameters are made public $(p, g, M)$

  where
  - $p$ is a prime
  - $g$ is the generator of a cyclic subgroup of $Z_p^*$
  - $M$ is the order of the cyclic subgroup generated by $g$

- It is important that the choice of $p$ and $g$ yield a large value for $M$
  - Although this is not sufficient; i.e. you should also avoid to have small factors of $p-1$
- A typical value for $g$ is 2



121

# FROM $Z_p^*$ TO DIFFIE-HELLMAN (2)

- Let *Xa* and *Xb* the private keys of entities A and B respectively
- Let *Ya* and *Yb* be the public keys of entities A and B respectively
- **Algorithm**
  - A selects a random number *Xa* such that *1<= Xa < M*
    - *Note*: if *g* generates the whole $Z_p^*$ then *Xa* is selected from {*2, …, p-2*} *since* $g^{p-1} \equiv 1 \ (mod\ p)$ by Fermat's Theorem in Group Theory
  - A computes $Ya = g^{Xa} \ mod\ p$
  - B selects a random number *Xb* such that *1<= Xb < M*
  - B computes $Yb = g^{Xb} \ mod\ p$
  - Upon reception of *Yb*, A computes the shared key $Ka = Yb^{Xa} mod\ p$
  - Upon reception of *Ya*, B computes the shared key $Kb = Ya^{Xb} mod\ p$
  - It is possible to show that *Ka=Kb* by simple mathematical manipulations…

$$
\begin{aligned}
&Ka = Yb^{Xa} mod\ p \\
&= \left(g^{Xb}\ mod\ p\right)^{Xa} mod\ p \\
&= \left(g^{Xb}\right)^{Xa} mod\ p = g^{Xb*Xa} mod\ p \\
&= \left(g^{Xa}\right)^{Xb} mod\ p \\
&= \left(g^{Xa}\ mod\ p\right)^{Xb} mod\ p = Ya^{Xb} mod\ p \\
&= Kb
\end{aligned}
$$



# FROM $Z_p^*$ TO DIFFIE-HELLMAN (3)

- **Example**
  - Consider *p=17:* $Z_p^* = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$
  - Consider now *g=2*: cyclic subgroup of order *M=8,* namely {1, 2, 4, 8, 16, 15, 13, 9}
    - Notice that *8* divides *17-1* according to Lagrange theorem
  - A chooses *Xa = 5* as a number between 1 and 8 as its private key
  - A's public key is *Ya = 2^Xa mod 17 = 2^5 mod 17 = 15*
  - B chooses *Xb = 7* as a number between 1 and 8 as its private key
  - B's public key is *Yb = 2^Xb mod 17 = 2^7 mod 17 = 9*
  - The secret session key as calculated by A: *Ka = Yb^Xa mod 17 = 9^5 mod 17 = 8*
  - The secret session key as calculated by B: *Kb = Ya^Xb mod 17 = 15^7 mod 17 = 8*

# SOME REMARKS

- The crucial property of the Diffie-Hellman algorithm is that an attacker having access to the public keys of both A and B is still not be able to figure out the shared key
- This is so also because parties A and B create a shared secret key without either party having to send it directly to the other (as it was the case with RSA)
- The Diffie-Hellman algorithm is also referred to as the **ephemeral secret key agreement protocol** because, typically, the shared secret key is used only once



# SECURITY OF DIFFIE-HELLMAN (1)

- Based on the fact that while it is relatively easy to compute the powers of an integer in a finite field, it is extremely hard to compute the discrete logarithms
- This means that while it is possible to quickly compute

$$Y = g^X mod\ p$$

  to determine the public key of an entity…
- … for an attacker, it is computationally infeasible to figure out the private key $X$ of the entity from the knowledge the public parameters p, g, M and the public key Y…
- … this is so because the attacker would need to perform the following discrete logarithm calculation

$$X = dlog_{g,p}\ Y$$

  for which **an efficient algorithm is not known**

Computational Diffie-Hellman assumption

# NOTE ON SOLVING THE DISCRETE LOGARITHM PROBLEM

Other methods exists but they are not much better than exponential in the number of bits required to represent the prime $p$

- Recall that $g^s \equiv k \ (mod\ p)$ and $dlog_{g,p}\ k = s$
- If an attacker can solve

$$g^s \equiv k \ (mod\ p)$$

with respect to $s$ for given values of $g$ and $k$, the security of the Diffie-Hellman algorithm is violated
  - This is the discrete logarithm problem
- An obvious way to solve this is by **brute force enumeration**
  - Compute $g^i\ mod\ p$ for increasing values of $i$ until a solution is found
  - The computational complexity of this is proportional to $p$
  - If $p$ represented in binary requires $n$ bits, then the complexity is proportional to $2\text{^}n$ and thus grows exponentially with the size of $p$ in bits



# SECURITY OF DIFFIE-HELLMAN (2)

- **Most serious vulnerability: man-in-the-middle attack**
- Assume an attacker *I* that can intercept all messages and decide to replay some of them in a possibly modified form
- The attack is as follows
  1. A→I : $g^a\ mod\ p$
  2. I→B : $g^{i_a}\ mod\ p$
  3. B→I : $g^b\ mod\ p$
  4. I→A : $g^{i_b}\ mod\ p$

Notice that the attacker is free to modify the content of the exchanged messages between A and B at will without being detected

  - A computes the shared key with whom it believes to be B but is I instead: $\text{Ka} = g^{a*i_b}\ mod\ p$
  - B computes the shared key with whom it believes to be A but is I instead: $Kb = g^{b*i_a}\ mod\ p$
  - I computes the same two shared keys: one with A and the other with B
  - I can decrypt the messages from A using *Ka* and re-encrypt them with *Kb* before forwarding to B and vice versa from the messages received from B and it is able to read all the exchanged messages!

# SECURITY OF DIFFIE-HELLMAN (3)

- The problem is that A and B are not authenticated
- To mitigate the problem, we need to include authentication in the Diffie-Hellman algorithm for key exchange
- This variant is referred to as **authenticated Diffie-Hellman** and it is organized as follows
  - each party acquires a certificate for the other party
  - the public key that each party sends to the other party is digitally signed by the sender using the private key that corresponds to the public key on the sender's certificate
  - this phase is usually performed by using RSA cryptography
- Authenticated Diffie-Hellman tries to combine the best of both Diffie-Hellman and RSA cryptography: it supports forward secrecy and avoids men-in-the-middle-attacks



# SECURITY OF DIFFIE-HELLMAN (4)

- For a complete overview of the problems concerning the Diffie-Hellman protocol, have a look at the following paper:

  https://www.researchgate.net/publication/2401745_Security_Issues_in_the_Diffie-Hellman_Key_Agreement_Protocol

- Recently, some observations about the procedures used to select the public parameters of the Diffie-Hellman algorithm have been put forward that may pave the way to large scale attacks
- **Key observation**: it may be computationally difficult to find the primes with the desirable properties for use with the Diffie-Hellman algorithm as they must yield multiplicative subgroups of large order and satisfy several other properties that are specified in the "*The OAKLEY Key Determination Protocol*" [RFC 2412]
  - https://tools.ietf.org/html/rfc2412

# SECURITY OF DIFFIE-HELLMAN (5)

- RFC 2412 recommends to use "safe" primes of
  - **length 768** bits such as *p=155251809230070893513091813125848175563133404943451431320235 119490296623994910210725866945387659164244291000768028886422 915080371891804634263272761303128298374438082089019628850917 069131659317536746955176311984337163722100721057791*9 with generator *g=2*
  - **length 1024 bits** such as *p=179769313486231590770839156793787453197860296048756011706444 423684197180216158519368947833795864925541502180565485980503 646440548199239100050792877003355816639229553136239076508735 759914822574862575007425302077447712589550957937778424442426 617334727629299387668709205606050270810842907692932019128194* with generator *g=2*
  - **length 1536 bits** …
- Security issues may arise since a **large number of servers use the same set of Diffie-Hellman parameters** following the recommendations of RFC 2412
- This reduces the cost of large-scale attacks, bringing some within range of feasibility today: an attacker can pre-compute for the recommended choices of the primes and solve the discrete logarithm problem to figure out the private from the public keys
- Nowadays, it seems possible to solve the discrete-log problem for 768-bit primes by academic researchers and for 1024-bit primes by state-level attackers



More on this problem at
https://weakdh.org/imperfect-forward-secrecy-ccs15.pdf




# OVERVIEW OF ECC

# INTRODUCTION (1)

- The computational overhead of the RSA-based approach to public-key cryptography increases with the size of the keys
- As algorithms for integer factorization scale up because of the availability of more and more computational power, RSA must resort to longer and longer keys
- **Elliptic curve cryptography (ECC)** can provide the same level and type of security as RSA (or Diffie-Hellman) with **much shorter keys**

| AES | RSA/Diffie-Hellman | ECC |
|---|---|---|
| 80 | 1024 | 160 |
| 112 | 2048 | 224 |
| 128 | 3072 | 256 |
| 192 | 7680 | 384 |
| 256 | 15360 | 512 |

**NOTE**
- For comparable key lengths, the computational efforts of RSA and ECC are comparable
- This implies that it takes ECC 1/6th of the computational effort to provide the same level of cryptographic security of 1024-bit RSA

- Comparable level of security against brute force attacks
  - AES: guessing the key
  - RSA: solving factorization pb
  - Diffie-Hellman: solving discrete log pb
  - ECC: … (we will see)



# INTRODUCTION (2)

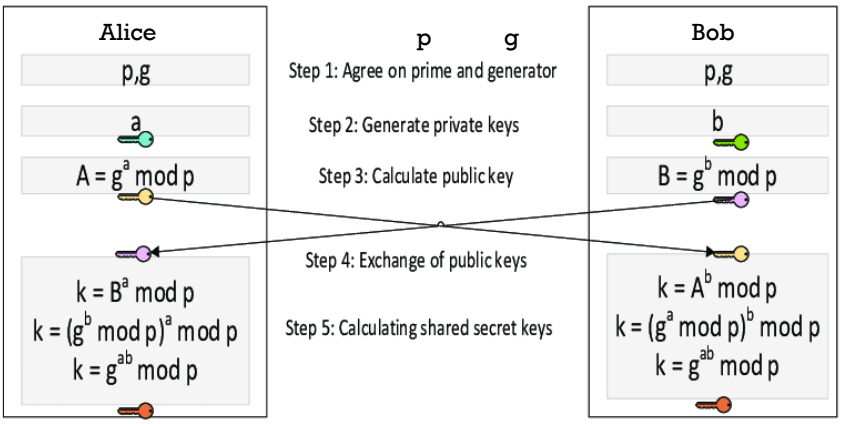
- The computational overhead of both RSA and ECC grows as *O(N^3)*
  - *N* is the key length in bits
- However, it takes far less computational overhead to use ECC on account of the fact that shorter keys in ECC provide the same level of security of much longer keys in RSA
- Other advantages deriving from shorter keys
  - ECC algorithms can be implemented on smartcards without mathematical coprocessors
  - Contactless smart cards work preferably with ECC because other algorithms require too much induction energy
  - Since shorter key lengths translate into faster handshaking protocols, ECC is also becoming increasingly important for wireless communications and more recent technologies like wireless sensors networks

# INTRODUCTION (3)

- It is possible to support forward secrecy and authentication, by using **ECDHE-RSA**
  - ECDHE = Elliptic Curve Diffie-Hellman Ephemeral
  - For SSL/TLS:
    - RSA is used for certificate based authentication
    - ECDHE is used for creating a one-time session key (in a way similar to DHE, more on this below)
- Note on the use of RSA for authentication
  - The main reason RSA is widely used for authentication is because a majority of the certificates in use today are based on RSA public keys
  - This is changing as more and more organizations use ECC based certificates
  - These new certificates use the ECDSA algorithm for authentication
  - When authentication is done by ECDSA and the session key generated with ECDH or ECDHE, the combined algorithm is denoted ECDHE-ECDSA or ECDH-ECDSA



# DIFFIE-HELLMAN KEY EXCHANGE



# RECALL THE MAIN IDEA UNDERLYING THE DIFFIE-HELLMAN KEY EXCHANGE

- Given the following two public :
  - $p$ is a prime
  - $g<p$ is a group generator
- The Diffie-Hellman algorithm for key exchange between two parties A and B is
  1. A→B : $g^a \bmod p$
  2. B→A : $g^b \bmod p$
- A can compute the (pre-)shared key with B as $(g^b \bmod p)^a$
- B can compute the (pre-)shared key with A as $(g^a \bmod p)^b$
- It turns out that $(g^b \bmod p)^a = g^{b*a} \bmod p = g^{a*b} \bmod p = (g^a \bmod p)^b$

- Observe that the **main operation** here is **to multiply the group generator *g* a certain number of times** (*a* and *b* respectively) modulo *p*…
- Recall also that the security of the algorithm is based on the fact that calculating *a* or *b* from $g^a \bmod p$ or $g^b \bmod p$ is infeasible
- **ECC is based on a generalization of the idea of repeatedly applying the group operation and the difficulty of computing how many times this has been applied**
- The main difference is that ECC uses points in a two dimensional space rather than integers…



# ECC: MAIN IDEA (1)

- Let *E* be a (finite) set of points on the plane
- Define a binary operation + on the points of *E* that satisfies the properties of a group operator (i.e. closure, associativity, identity and invertibility)
  - Given points *P* and *Q* in *E*, then *P*+*Q*=*R* in *E*
- Because of the observations in the previous slide, we are interested in considering a point *G* and compute
  - *G*+*G*
  - *G*+*G*+*G*
  - *G*+*G*+*G*+*G*
  - …

  i.e. *k*-times addition of the point *G* that will be also written as *k***G*

# ECC: MAIN IDEA (2)

- The crucial property that the set *E* needs to satisfy to be useful for cryptography in a sense similar to the Diffie-Hellman is that

  **after we have calculated *k* * *G* for a given point *G* in *E*,**

  **it is extremely difficult to recover *k* from *k* * *G***

- This means that we assume that the only way to recover *k* from *k* * *G* is to try every possible repeated summation like
  - *G* + *G*
  - *G* + *G* + *G*
  - …

  until we get the same point *k* * *G*

- Trying to figure out the value k in *k**G is called the **discrete logarithm problem**
- This becomes apparent if we replace the group operator + with * and thus the *k*-fold application of + to *G* can be written as *G^k*…
- … thus finding *k* becomes the inverse of exponentiation, i.e. computing the logarithm



# ECC: MAIN IDEA (3)

- ECC based Diffie-Hellman key exchange algorithm
- Make the point *G* public
- A selects an integer *Xa* as private key
  - A's public key is set to *Ya* = *Xa* * *G*
    - This means that *Ya* is obtained by the *Xa*-fold application of the operation + to the point *G*
    - This implies that **while the private key is an integer, the public key is a point**
- B selects an integer *Xb* as private key
  - A's public key is set to *Yb* = *Xb* * *G*
    - This means that *Yb* is obtained by the *Xb*-fold application of the operation + to the point *G*
    - This implies that while the private key is an integer, the public key is a point
- A and B exchange their public keys *Ya* and *Yb*…

# ECC: MAIN IDEA (4)

- A and B exchange their public keys *Ya* and *Yb*
- A computes the shared key $Ka = Xa * Yb = Xa * (Xb * G) = (Xa*Xb) * G$
- B computes the shared key $Kb = Xb * Ya = Xb * (Xa * G) = (Xb*Xa) * G$
- Indeed, we have that $Ka = Kb$

- **Key observation**

  *All of the assumptions about the set E of points are satisfied when the points in E are taken from an **elliptic curve***



# A REMARK (1)

- Before introducing elliptic curves, let us remark that if the security of ECC depends on finding out how many times a point *G* participates in a sum like $G + G + \ldots + G$, why would it take an attacker any more work to figure that out than it would take for a party to calculate the sum? It would seem that all that the attacker would need to do would be to keep on adding *G* to itself until the attacker obtains the value of the sum
- In other words, if some integer *Xa* is the private key of A and if A derives its public key by adding the point *G* to itself *Xa* times, the amount of computational effort A expends in adding *G* to itself *Xa* times should be the same as what the attacker would need to expend if it kept on adding *G* to itself until reaching a value that is A's public key

# A REMARK (2)

- To understand why it is not the case, the amount of computational effort that it takes to add a point *G* to itself *Xa* number of times is logarithmic in the size of *Xa*
- This is so because
  - First, A adds *G* to itself once and gets *2 * G*
  - Next, A adds *2 * G* to itself and gets *4 * G*
  - *Then,* A adds *4 * G* to itself, it gets *8 * G*
  - … and so on
- However, the attacker would not know the value of *Xa* and so it would not be able to take advantage of such exponentially increasing jumps
- Additionally, in most commonly used ECCs, all the calculations are carried out modulo a prime *p*
  - Thus, as the attacker keeps on adding *G* to itself, the size of what it gets cannot serve as a guide to how many more times the attacker must repeat that addition to get to the final value



143

# ELLIPTIC CURVES OVER THE REALS

# WHY ELLIPTIC?

A **ring** is an abelian group with a second binary operation (*) that is associative, distributive over the abelian group operation (+), and has an identity element (1)

- **Elliptic curves** are called so because of their **relationship to elliptic integrals**
- An elliptic integral can be used to determine the arc length of an ellipse
- So, elliptic curves have nothing to do with ellipses
  - Ellipses are described by quadratic curves while elliptic curves are always cubic

Weierstrass equation of characteristic 0

$$y^2 = x^3 + ax + b$$

- The meaning of "Weierstrass equation of characteristic 0" is that **the set of numbers that satisfy the equation constitutes a ring of characteristic 0**
- The characteristic of a ring is the number of times one must add the multiplicative identity element in order to get the additive identity element
- A characteristic 0 means that no matter how many times one adds the multiplicative identity element, it is impossible to get the additive identity element

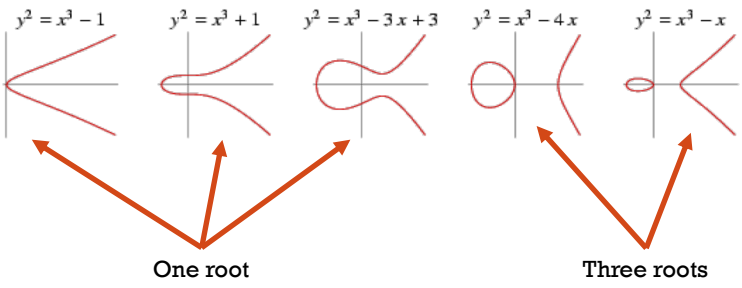


# SHAPES OF ELLIPTIC CURVES

- Considering the cubic polynomial in the Weirstrass equation, it is possible to compute its determinant and study when it has one, two or three roots

Examples

Notice that elliptic curves are symmetric in the x-axis as it is clear whenever we rewrite the Weierstrass equation as $y = \pm\sqrt{x^3 + a * x + b}$



# AN ELLIPTIC CURVE USED IN PRACTICE

$$y^2 = x^3 + 3176890812513255034763174764138276932727469559 27 * x + 79052896607878758718120572025718535432100651934$$

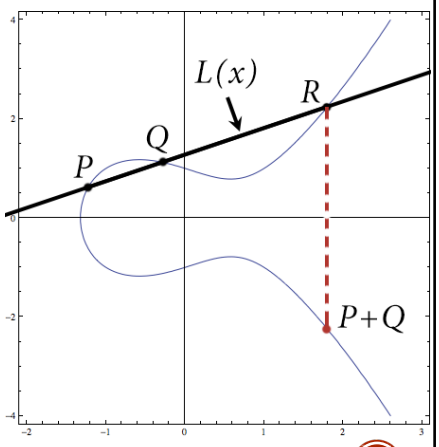
This curve is used for **digital rights management (DRM)**

- DRM is a set of access control technologies for restricting the use of proprietary hardware and copyrighted works
- DRM technologies try to control the use, modification, and distribution of copyrighted works (such as software and multimedia content), as well as systems within devices that enforce these policies



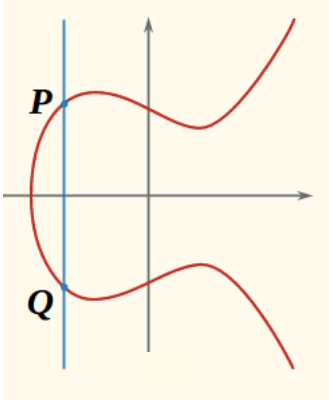
# FEATURES OF ELLIPTIC CURVES (1)

- The points on an elliptic curve can be shown to constitute a group
  - A group operator; an identity element with respect to the operator; closure and associativity with respect to the operator; existence of inverses w.r.t. the operator
- The **group operator for the points on an elliptic curve** is called **addition** although its definition has nothing to do with the conventional arithmetic addition
- The definition of two points *P* and *Q* (denoted *P*+*Q*) is the following
  - join P with Q with a straight line
  - call the third point *R* of the intersection of the straight line through *P* and *Q* with the curve, **if such an intersection exists**
  - the mirror image of *R* with respect to the *x*-axis is the point *P* + *Q*
  - …



# FEATURES OF ELLIPTIC CURVES (2)

- The definition of two points $P$ and $Q$ (denoted $P+Q$) is the following
  - …
  - **If the third point of intersection does not exist, we say it is at infinity**
  - We denote the point at infinity by the special symbol $O$ that will be defined as the additive identity element for the group operator
  - The intuition for this choice is the following
    - $O$ is a point at infinity on the y-axis
    - Since the graph of the elliptic curve is symmetric w.r.t. the x-axis, the only way to have only two points intersecting the curve is to have the third at infinity on the y-axis

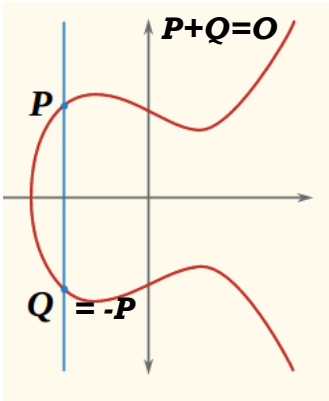





# FEATURES OF ELLIPTIC CURVES (3)

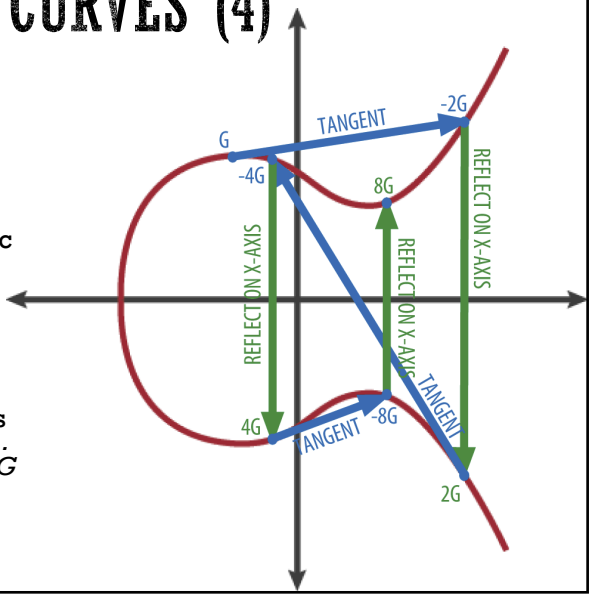
**Some properties of O**

- $P+O=P$ for any point $P$
- By definition, the additive inverse of a point $P$ is its reflection with respect to the x-axis
  - If $Q$ is the mirror reflection of $P$ on the curve, then $Q = -P$
  - The point of intersection of a point and its additive inverse will be the distinguished point $O$
  - This implies that $O = -O$



# FEATURES OF ELLIPTIC CURVES (4)

- What does it mean to add a point *G* to itself?
- To answer the question take *G* and a distinct point *H*, then move *H* closer and closer to *G*
- In the limit, *H* will overlap with *G* and the line joining *G* and *H* will be the tangent to the elliptic curve in *G*
- So, *G* + *G* is defined as finding a tangent at *G*, then determine the intersection of the tangent with the curve, and then take the reflection of such intersection with respect to the x-axis
- See, on the right the repeated application of this operation that allows one to obtain *2G, 4G, 8G, …* *where nG* abbreviates the repeated addition of *G* *n*-times



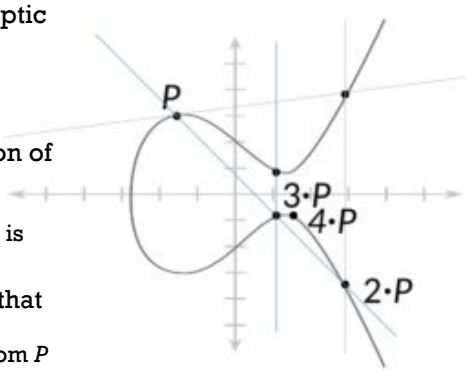



# FEATURES OF ELLIPTIC CURVES (5)

- *E(a,b)* denotes the set of points belonging to the elliptic curve described by

$$y^2 = x^3 + a * x + b$$

  extended with the point *O* at infinity
- *E(a,b)* can be shown to be closed under the operation of addition as defined above, that it has an inverse for every element, and the addition is associative
  - In other words, *E(a,b)* is a group (it can be shown that it is an Abelian group, i.e. addition is also commutative)
- It is possible to define repeated addition for values that are not power of 2, i.e. for *kP* where *k= 2, 3, 4, …*
  - Start with the tangent in *P* to get *2P*, the consider line from *P* to *2P* to get *3P*, then line from *P* to *3P* to get *4P*, …



# FEATURES OF ELLIPTIC CURVES (6)

- It is easy to find analytic expressions for the values of the coordinates of adding two points *P(xP,yP)* and *Q(xQ,yQ)*
- By using standard expressions for lines and basic properties of the cubic polynomials, one can derive that the coordinates of *P+Q* are

$$x_{P+Q} = \alpha^2 - x_P - x_Q$$
$$y_{P+Q} = \alpha(x_P - x_R) - y_P$$

where

$$\alpha = \frac{y_Q - y_P}{x_Q - x_P} \qquad x_R = \alpha^2 - x_P - x_Q$$



# FEATURES OF ELLIPTIC CURVES (7)

- Similarly, it is easy to find the analytic expressions of the values of the coordinates of *2P* for *P(xP,yP)*:

$$x_{2P} = \alpha^2 - 2x_P$$
$$y_{2P} = \alpha(x_P - x_R) - y_P$$

where

$$\alpha = \frac{3x_P^2 + a}{2y_P} \qquad x_R = \alpha^2 - 2x_P$$

- Notice the similarity (apart from the expression of $\alpha$) with those in the previous slide

154

# ELLIPTIC CURVES OVER FINITE FIELDS

For $p$ a prime



# INTRODUCTION (1)

- The elliptic curve operations considered above assume computations to be performed over real numbers
- This kind of arithmetic computations cannot be used for cryptography because calculations with real numbers are prone to round-off error whereas **cryptography requires error-free arithmetic**
- The solution to this problem is to restrict the values of the parameters $a$ and $b$, the value of the independent variable $x$, and the value of the dependent variable $y$ to some finite field $Z_p$ for $p$ a prime number
- In other words, we consider elliptic curves that satisfy the following congruence

$$y^2 \equiv x^3 + a * x + b \ \ (mod\ p)$$

with the proviso that $4 * a^3 + 27 * b^2 \neq 0\ (mod\ p)$

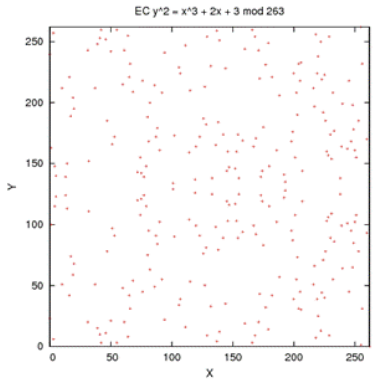
  - The proviso aims to rule out curves with singularities that substantially decrease the security of the cipher using the elliptic curve



155

# INTRODUCTION (2)

- This is how an elliptic curve over a finite field looks like
- All the points belonging to the curve have integer coordinates






# ELLIPTIC CURVES OVER $Z_p$

- Similarly to what we have done for elliptic curves over the reals, we denote by $E_p(a, b)$ the set of points that satisfy

$$y^2 \equiv x^3 + a * x + b \ \ (mod\ p)$$

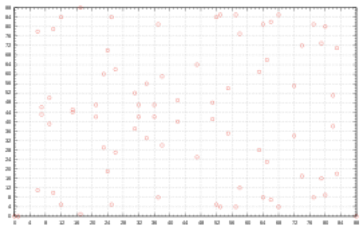
  with the proviso that $4 * a^3 + 27 * b^2 \neq 0\ (mod\ p)$
- Indeed, $E_p(a, b)$ is no longer the set of real points belonging to a curve, but a collection of discrete points in the Cartesian product $Z_p \times Z_p$
- Since the points in $E_p(a, b)$ no longer form a curve, we cannot use the geometrical construction to illustrate the action of the addition operator
- Luckily, **the algebraic expressions derived for these operations continue to hold good provided the calculations are carried out modulo *p***
- The set $E_p(a, b)$ of points, with the elliptic curve defined over $Z_p$, is a group

# ELLIPTIC CURVES OVER FINITE FIELDS

- In general, it is possible to define elliptic curves over any finite field
- It possible to count the number of points in an elliptic curve over a finite field by using **Schoof's algorithm**
  - It is a deterministic polynomial time algorithm
- Knowing the number of points is very important for judging the difficulty of solving the discrete logarithm problem in the group of points of the elliptic curve
- **Hasse theorem** provides an estimate of the number of points on an elliptic curve over a finite field, bounding the value both above and below:

$$|N - (q+1)| \leq 2\sqrt{q}$$

where N is the number of points and $q$ is the number of elements in the finite field



Set of affine points of elliptic curve $y^2 = x^3 - x$ over finite field $Z_{89}$



# ELLIPTIC CURVES OVER GALOIS FIELDS (1)

- Of particular interest, it is the definition of elliptic curves over Galois fields, i.e. over $GF(2^n)$
- For these curves to be cryptographically secure, it is mandatory to take ***n* a prime**
- Recall that over Galois fields, addition is equivalent to bit-wise xor
- This implies that $x+x=0$ that in turn implies that the field has characteristic 2
- This prevents the use of the class of elliptic curves described by the congruence

$$y^2 \equiv x^3 + a * x + b \pmod{p}$$

**since they become singular and this is detrimental to security**

- The problem is related to the fact that it may become impossible to compute the addition of a point by itself (remember the geometric construction that requires to identify the tangent to the elliptic curve which is not possible when the point is a singularity)

# ELLIPTIC CURVES OVER GALOIS FIELDS (2)

- The elliptic curve equation to use when the underlying field is described by $GF(2^n)$ is the following

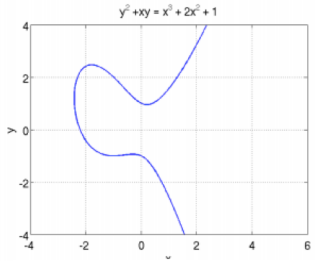
$$y^2 + x * y = x^3 + a * x + b$$

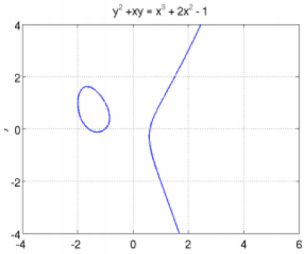
where all operations are carried out in $GF(2^n)$ with the proviso that

$$b \neq 0$$

to avoid singularities (that makes drawing tangent impossible and thus also computing the addition of a point to itself)

- Notice that $b = 0$ is sufficient for having singular elliptic curves but it may happen that an elliptic curve is singular although $b \neq 0$






# ELLIPTIC CURVES OVER GALOIS FIELDS (3)

- The fact that the expression for describing elliptic curves over $GF(2^n)$ is different from the one over the reals or over finite fields requires some modifications to the expressions for adding points…
- Given a point *P = (x, y)*, the reflection over the x-axis is defined as

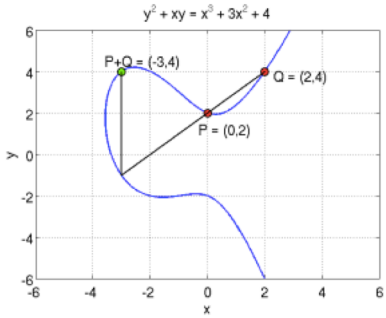
$$-P = (x, -(x + y))$$

- Adding two points *P=(xP,yP)* and *Q=(xQ,yQ)* is defined as

$$x_{P+Q} = \alpha^2 + \alpha - x_P - x_Q - a$$
$$y_{P+Q} = -\alpha(x_{P+Q} - x_P) - x_{P+Q} - y_P$$

with

$$\alpha = \frac{y_Q - y_P}{x_Q - x_P}$$



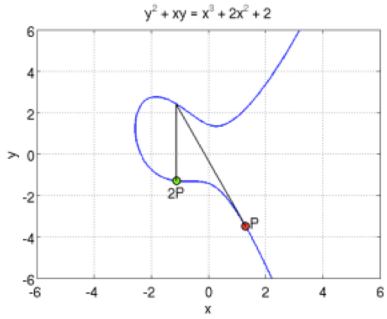
# ELLIPTIC CURVES OVER GALOIS FIELDS (4)

- Adding a point *P=(xP,yP)* to itself, i.e. computing *2P*, is defined as

$$x_{2P} = \alpha^2 + \alpha - a - 2x_P$$
$$y_{2P} = -\alpha^2 - \alpha + a + (2+\alpha)x_P - \alpha x_{2P} - y_P$$

with

$$\alpha = \frac{3{x_P}^2 + 2ax_P - y_P}{2y_P + x_P}$$






# ELLIPTIC CURVES OVER GALOIS FIELDS (5)

- If $g$ is a generator for $GF(2^n)$, then all its elements can be expressed as $0,1,g,g^2,\dots,g^{2^n-2}$
- This means that the points in the elliptic curve $E_{2^n}(a,b)$ will be
  - $O$
  - $(0,g^i)$ for some $i$
  - $(g^i,0)$ for some $i$
  - $(g^i,g^j)$ for some $i,j$

A generator $g$ of a finite field is such that each non-zero element of the field can be expressed as $g^i$ for some value of $i$

- For security (i.e. solving the discrete logarithm problem hard), the **order** of $E_{2^n}(a,b)$, that is the number of points in the curve, is crucial
- By recalling Hasse theorem and the fact that $GF(2^n)$ contains $2^n$ elements, we can derive that $E_{2^n}(a,b)$ contains $2^n + 1 - t$ for $t \leq 2^{\frac{n}{2}}$

164

# ELLIPTIC CURVES CRYPTOGRAPHY (ECC)



## ECC: IDEA (1)

- In 1985, Neal Koblitz (University of Washington) and Victor Miller (IBM) put forward the idea that elliptic curves over finite fields can be used in cryptography
- Just as RSA uses exponentiation, i.e. repeated multiplication, as its basic arithmetic operation, ECC uses the addition group operator as its basic arithmetic operation
- Assume $G$ is a point, chosen by a user, over an elliptic curve $E_q(a, b)$ where
  - $q$ is a prime (finite field over a prime) or
  - $q = 2^n$ for $n$ a prime (Galois field)
- The key insight underlying ECC is that

**with an appropriate choice for $G$, whereas it is relatively easy to calculate $C = M * G$**

**for an integer $M$ encoding a plaintext represented as a sequence of bits, it is infeasible to derive $M$ from $C$ even in the case of knowing $G$ and the curve $E_q(a, b)$**

Deriving $M$ from $C$ is called the discrete logarithm problem



165

# ECC: IDEA (2)

- An attacker could try to recover *M* from $C = M * G$ by calculating *2*G, 3*G, 4*G, . . .*, *k*G* with increasing values of *k*
- In the worst case, the process will span all points in the elliptic curve $E_q(a, b)$ and requires to check whether or not the result is equal to *C*
- Indeed, if *q* is sufficiently large and the point *G* on the curve $E_q(a, b)$ is chosen carefully, this would take too long in practice and it can thus be considered as infeasible
- ECC is typically not used for encryption but rather for key exchange in a similar way as it was done in the Diffie-Hellman key exchange algorithm



# ECDH: EC DIFFIE-HELLMAN (1)

- The goal is to establish a secret session key between two parties
- A community of users wishing to engage in secure communications with ECC chooses the parameters *q*, *a*, and *b* for an elliptic curve $E_q(a, b)$ together with a base point *G* in $E_q(a, b)$
  - Recall that *q* can either be a prime or a power of 2
- **Algorithm ECDH**
  - A selects an integer *Xa* as its private key and compute $Ya = Xa * G$ as its public key
  - B selects an integer *Xb* as its private key and compute $Yb = Xb * G$ as its public key
  - A and B exchanges their public keys
  - A computes $Ka = Xa * Yb$ and B computes $Kb = Xb * Ya$
- It turns out that $Ka = Kb$

# ECDH: EC DIFFIE-HELLMAN (2)

- To discover the secret session key, an attacker could try to discover *Xa* from the publicly available base point *G* and the public key *Ya* since $Ya = Xa * G$
- As already argued this amounts to solving the discrete logarithm problem that can be very hard to tackle by choosing a suitable elliptic curve and a point *G*
- For the point *G*, a good strategy is to select points over the elliptic curve whose order is very large where **order** is the **least number of times *G* must be added to itself so that we get the identity element *O***
- The base point *G* can also be seen as the generator of a subgroup of $E_q(a, b)$ whose elements are *O, G, 2G, 3G, . . .*
  - In case $q = 2^n$ and the finite field is a Galois field, for the size of the subgroup to equal the degree of the generator *G*, the value of *n* must be a prime



# ECDSA: EC DIGITAL SIGNATURE ALGORITHM (1)

- Consider a finite field $Z_p$ over a large prime *p*
  - We do not consider Galois fields
- Select the parameters *a* and *b* for the curve and a generator point *G* of high order *n* (meaning that *n* * *G* = *O* for a large value of *n*)
- **Algorithm for signing**
  - A randomly selects *X* such that $1 \leq X \leq n - 1$ as its private key
  - A calculates its public key $Y = X * G$
  - A makes *p, a, b, G, n* and *Y* publicly available while keeping *X* secret
  - A generates a random number *K* such that $0 < K < n - 1$
    - This number is one-time as it will be used only once, i.e. any two signatures will be computed by using distinct random values for *K*
  - Let *H* be a digest (generated, e.g., by a hash function) of the document that A wants to sign

# ECDSA: EC DIGITAL SIGNATURE ALGORITHM (2)

Taking only the x-coordinate of the point $K*G$

This is the inverse of $K$ modulo $n$

- **Algorithm for signing(continued)**
  - A computes
    - $sig1 = (K * G)_x \ mod \ n$
    - $sig2 = K^{-1}(H + X\ sig1) \ mod \ n$
  - A sends *(sig1,sig2)* together with the document to the recipient B
- **Algorithm for signature verification**
  - B calculates the digest of the received document (by, e.g., using the same hash function)
  - B computes the following numbers
    - $w = sig2^{-1} \ mod \ n$
    - $u_1 = H\ w \ mod \ n$
    - $u_2 = sig1\ w \ mod \ n$
  - B checks if $sig1 \equiv (u_1 * G + u_2 * Y)_x \ (mod \ n)$

For details and proof of the ECDSA algorithm, please look at https://link.springer.com/article/10.1007/s102070100002



171

# SECURITY OF ECC

# REMARKS (1)

- Just as RSA depends on the difficulty of large-number factorization for its security, ECC depends on the difficulty of large numbers discrete logarithm calculation
- It was shown by Menezes, Okamoto, and Vanstone (**MOV**) in 1993 that (for **singular** or **supersingular** elliptic curves) the problem of solving the discrete logarithm problem can be reduced to the much easier problem of finding logarithms in a finite field
- To avoid the MOV attack, the underlying elliptic curve and the selected base point must satisfy what is known as the **MOV Condition**, i.e.

**the order *m* of the base-point should not divide $q^B - 1$ for small *B*, say for *B* < *20***

  - Notice that $q$ is the prime $p$ when the underlying finite field is $Z_p$ or it is $2^n$ when the underlying finite field is the Galois field $GF(2^n)$



# REMARKS (2)

- When using Galois fields $GF(2^n)$, another security consideration relates to what is known as the **Weil descent attack**
- To avoid this attack, $n$ must be a prime

- Elliptic curves for which the total number of points on the curve equals the number of elements in the underlying finite field are also considered cryptographically weak

# HOW TO CHOOSE AN ELLIPTIC CURVE?

- Several standards available such as

  https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

- Each of the standards tries to ensure that the **elliptic-curve discrete-logarithm problem** (ECDLP) is difficult
- Unfortunately, there is a gap between ECDLP difficulty and ECC security
- None of the available standards do a good job of ensuring ECC security since there are many attacks that break real-world ECC without solving ECDLP!
- The core problem is that implementations fail with respect to several different aspects such as wrong results on some rare curve points, leaks secret data when the input is not a curve point, …
- These problems are exploitable by real attackers, taking advantage of the gaps between ECDLP and real-world ECC:
  - ECDLP reveals only $k*P$ whereas real-world ECC also reveals timing
  - ECDLP always computes $n*P$ correctly. Real-world ECC has failure cases.
- Secure implementations of the standard curves are theoretically possible but very hard



To have an idea of what can go wrong with ECC, read
https://fahrplan.events.ccc.de/congress/2010/Fahrplan/events/4087.en.html