

PV181 Laboratory of security and applied cryptography



**Part 3, seminar 10:
Advanced Topics**

Łukasz Chmielewski
chmiel@fi.muni.cz



Outline

- Organizational
- Finishing Seminar's 3 exercises (10min)
 - Discuss the most common issue in the solutions
 - Optional task discussion
- Recall from the last seminars
 - We concentrate on signatures
- RSA, RSA-CRT
- DSA, ECDSA
- Efficiency
- Post Quantum

Organization

- 22.11 Advanced Crypto by Lukasz +HW
- 29.11: Biometrics by Agata & Katarina +HW
- 6.12: Crypto-libraries protected against hardware attacks by Lukasz +HW
- 13.12: Passwords Security by Alessia and Lukasz +HW

Issues in Solutions

- Multiple OS, libraries, OpenSSL versions
 - In some cases even if you tell me the details, I cannot run it easily. Let's double-check the code together.
- Encryption of large files:
 - Encrypting line by line; line length might be very long
 - Encrypting chunks of files separately
 - they can be re-ordered
 - Forgetting to run finalize after encryption.

Finishing Seminar 3 last tasks

- demo.py

```
#Task 2: Encrypt the message below with a padding of your choice and the public key
str = b"a secret message"
#sys.exit()

#Task 3: Decrypt the message below with the private key
str = b"a secret message"
#sys.exit()

#Task 4: If there is time try to sign str

#Task 5: If there is time try to verify str

#Task 6 (optional): assume that you have public_key and d. Compute p and q.

#Task 7 (optional and harder) assume that you have public_key and dmp1. Compute d, p, and q.
```

- Tasks 6 and 7...
 - Did anyone try to do it?

Tasks 6 & 7:

#Task 6 (optional): assume that you have `public_key` and `d`. Compute `p` and `q`.

#Task 7 (optional and harder) assume that you have `public_key` and `dmp1`. Compute `d`, `p`, and `q`.

- Both are doable but mathematically not trivial for arbitrary e .
 - For a complete solution see: Handbook of Applied Cryptography (chapter 8, section 8.2.2, page 287). Link: <https://cacr.uwaterloo.ca/hac/>
- However e is normally small: $e=0x010001=65537$
- Therefore, the following attack is possible for d , n , and e known:
 1. We have $e \cdot d = k \cdot \varphi(n) + 1$ for some integer k . Since $d < \varphi(n)$, we know that $k < e$. So you only have a small number of k to try to get $\varphi(n)$.
 - Note that $\varphi(n) \approx n$ or at least most significant bits.
 - For small e : $k' = \text{round}(ed/n)$. k' is very close to k (i.e. $|k' - k| \leq 1$).
 2. Given k , you easily get $\varphi(n) = (ed - 1)/k$.
 3. Now $\varphi(n) = (p - 1)(q - 1) = n + 1 - (p + q)$. Thus, you get $p + q = n + 1 - \varphi(n)$.
 - So we have $p + q$ and $n = p \cdot q$.
 - Note that for all real numbers a and b , a and b are the two solutions of the quadratic equation $X^2 - (a + b) \cdot X + a \cdot b = 0$.
 4. Compute: ...

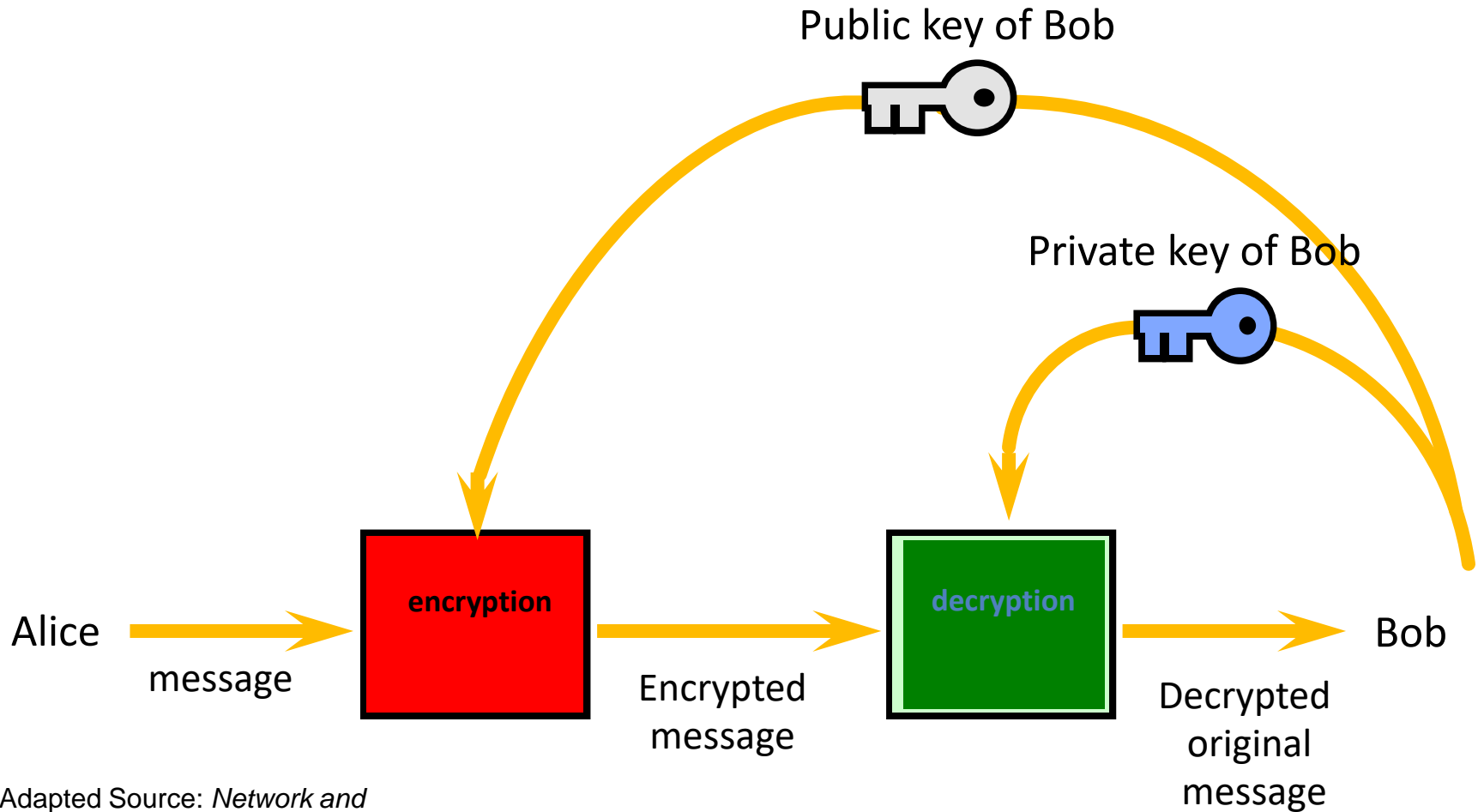
Tasks 6 & 7 cont'd

4. Compute:

- $p = ((p+q) + \sqrt{(p+q)^2 - 4 \cdot pq}) / 2$
- $q = ((p+q) - \sqrt{(p+q)^2 - 4 \cdot pq}) / 2$

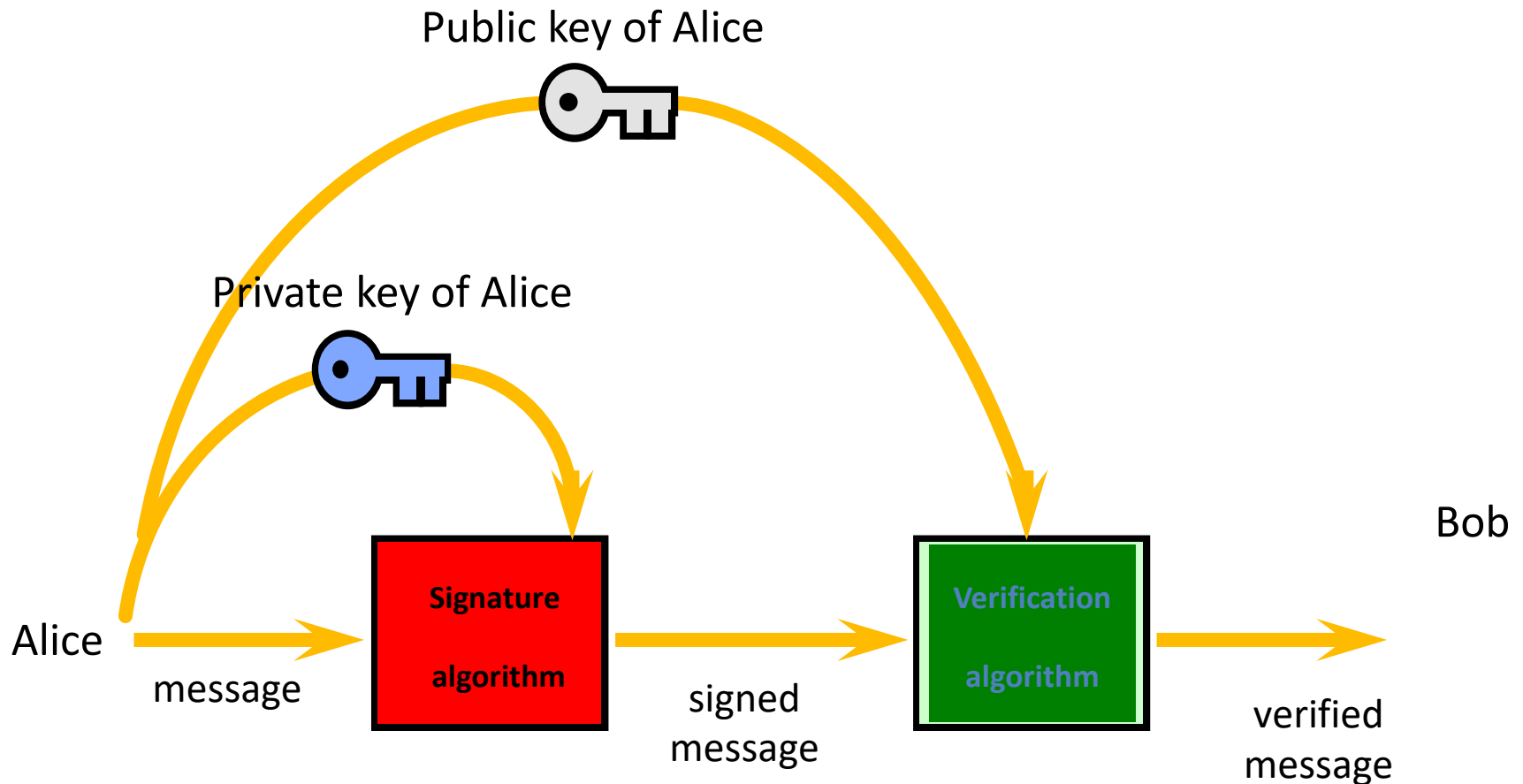
- Easier way, there is a function to compute p and q from (n,e,d):
 - `cryptography.hazmat.primitives.asymmetric.rsa.rsa_recover_prime_factors(n,e,d)`
- For $d_p = d \pmod{p}$, n, and e known, the attack is easier:
 - $d_p \leq d < \varphi(n)$
 - $e \cdot d_p = 1 \pmod{p-1}$ and therefore $e \cdot d_p = k \cdot (p-1)$
 - $k < e$ so we can brute force k again and we compute $x = e \cdot d_p + k = k \cdot p$
 - Now we compute $\gcd(x,n)$ to obtain p.
- More to read:
 - “Reconstructing RSA Private Keys from Random Key Bits”:
<https://eprint.iacr.org/2008/510.pdf>
 - “Weaknesses in Current RSA Signature Schemes”:
https://link.springer.com/chapter/10.1007/978-3-642-31912-9_11

Recall: Asymmetric cryptosystem



Adapted Source: *Network and Internet Security* (Stallings)

Digital signature scheme



Source: *Network and
Internetwork Security* (Stallings)

Digital signature



- Alice generates key pair
 - Public key is published (sent to Bob) for verification of signature
- Alice sign a document using her private key
- Bob use public key to verify the digital signature
- Classical examples: RSA, ECC
- Post-Quantum Schemes:
 - Why?

RSA: reminder

1. Secret primes p, q : $n = p \cdot q$
 2. Public exponent e :
 $\gcd(e, (p - 1)) = \gcd(e, (q - 1)) = 1$
 3. Private exponent d : $d \cdot e \equiv 1 \pmod{\varphi(n)}$
- Encryption (public n, e): $E(m) = m^e \pmod n = c$
- Decryption (private n, d): $D(c) = c^d \pmod n = m$



RSA-CRT: mathematics

- Optimization of computing a signature giving about 3 or 4-fold speed-up
- Precompute the following values:
 - Find $d_p = d \pmod{p-1}$, computed as $d_p = e^{-1} \pmod{p-1}$
 - Find $d_q = d \pmod{q-1}$
 - Compute $i_q = q^{-1} \pmod{p}$
- Computations using $m_p = m \pmod{p}$ and $m_q = m \pmod{q}$
- Signature or encryption (forgetting about hashing):
 - $s_p = m^{d_p} \pmod{p}$ 
 - $s_q = m^{d_q} \pmod{q}$ 
 - Garner's method (1965) to recombine s_p and s_q :
 - $s = s_q + q \cdot (i_q(s_p - s_q) \pmod{p})$

Discrete Logarithm Problem Z_p^* (or Z_n^*)

For g, p, y find integer x such that

$$y \equiv g^x \pmod{p}$$

$$g = 2, p = 31$$



$$g^x \pmod{p}: 2, 4, 8, 16, 1 = 2^5 \pmod{31} \text{ (order} = 5\text{)}$$

$$g = 3, p = 31$$

$$g^x \pmod{p}: 3, 9, 27, 19, \dots, 3^{30} \pmod{31} = 1$$

full order = 30, 3 is generator (all numbers)

DSA: reminder

- Signature generation
 - Generate a random per-message value k such that $0 < k < q$.
 - Calculate $r = (g^k \bmod p) \bmod q$ 
 - Calculate $s = (k^{-1}(H(m) + x*r)) \bmod q$ 
 - The signature is (r, s) .
- Signature verification
 - $w = (s)^{-1} \bmod q$
 - $u1 = (H(m)*w) \bmod q$
 - $u2 = (r*w) \bmod q$
 - $v = ((g^{u1}*y^{u2}) \bmod p) \bmod q$
 - The signature is valid if $v = r$
- For DSA (1024,160) the signature size will be 2x160 bits.

DSA: Padding

- Decide on lengths **L** and **N**, e.g. (1024,160).
 - N must be less than or equal to the hash output length
 - E.g. for (1024,160) sha-1 used to be used, sha-256 would be ok as well and only the first 160 bits would be used; nowadays (2048, 256) or (3072, 256) should be used;
 - $s = (k^{-1}(\mathbf{H}(\mathbf{m}) + x*r)) \bmod q$
- “It is recommended that the security strength of the (L, N) pair and the security strength of the hash function used for the generation of digital signatures be the same unless an agreement has been made between participating entities to use a stronger hash function. When the length of the output of the hash function is greater than N (i.e., the bit length of q), then the leftmost N bits of the hash function output block shall be used in any calculation using the hash function output during the generation or verification of a digital signature. A hash function that provides a lower security strength than the (L, N) pair ordinarily should not be used, since this would reduce the security strength of the digital signature process to a level no greater than that provided by the hash function.” [FIPS 186-3]

RSA, RSA-CRT, DSA problems

- Plus: no polynomial algorithm to solve factorization or discrete logarithm in \mathbf{Z}_p^* and \mathbf{Z}_n^*
- Minus: there exist algorithms for factoring and discrete log that are faster than a generic algorithm:
$$\exp\left((C + o(1))(\log n)^{1/3} (\log \log n)^{2/3}\right)$$
- A generic discrete log solver works in proportional time to the square root of the group size, and thus exponential in half the number of digits in the group.
- Result: bigger keys
- Solution: Elliptic Curves DSA

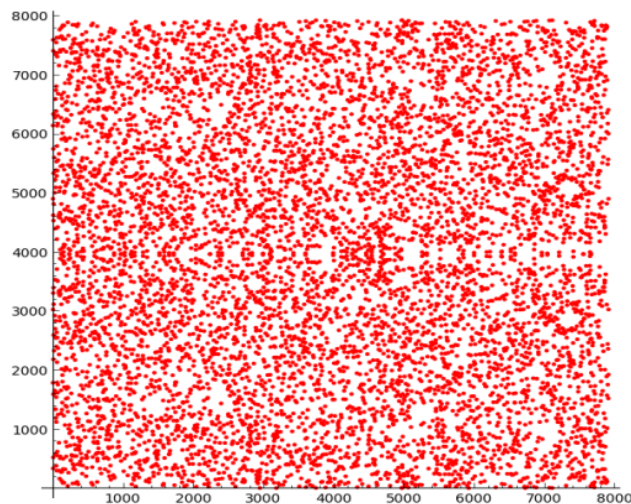
Elliptic curve DSA (ECDSA)

- Elliptic curves invented by Koblitz & Miller in 1985.
- ECDSA proposed in 1992 by Vanstone
- Became ISO standard (ISO 14888-3) in 1998
- Became ANSI standard (ANSI X9.62) in 1999

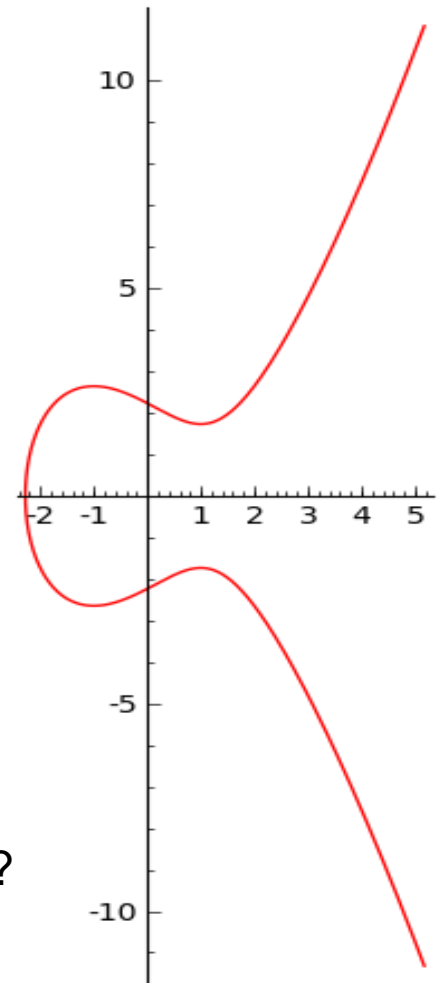
- ECDSA is a version of DSA based on elliptic curves.

Elliptic curve example

- Example
 - $y^2 = x^3 - 3x^2 + 5$ over \mathbb{Q} , and ∞
- How would it look over a finite field,
 - for example: F_p ? for $p = 7919$

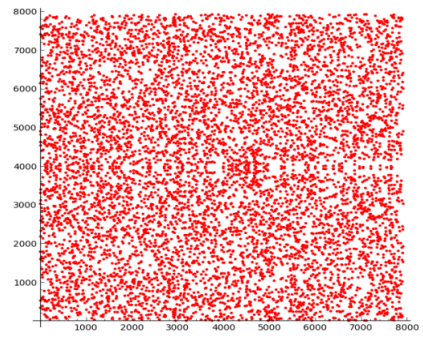
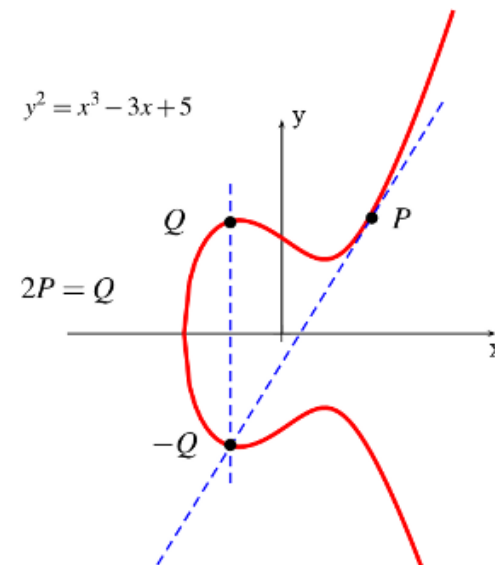
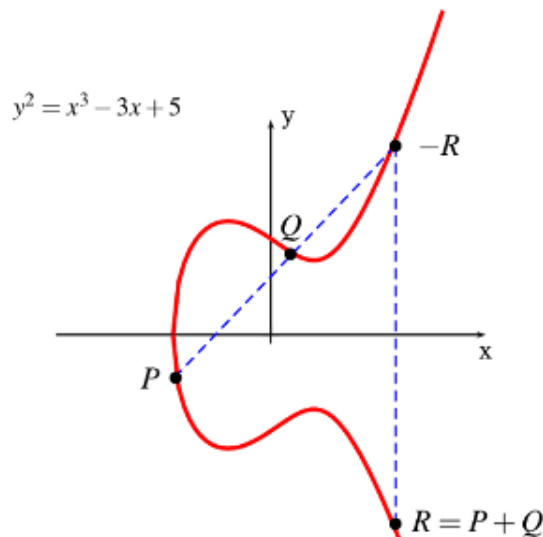


Can you see a pattern?



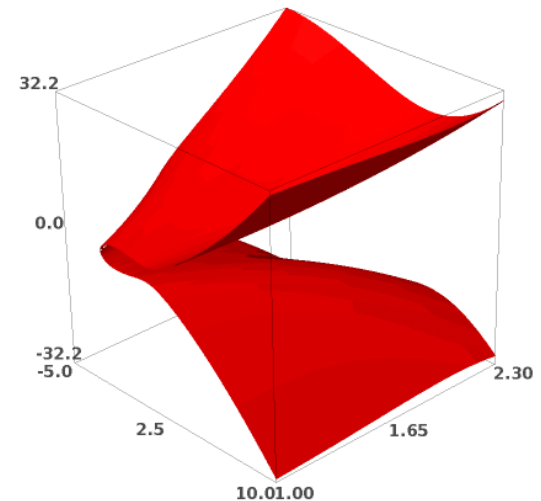
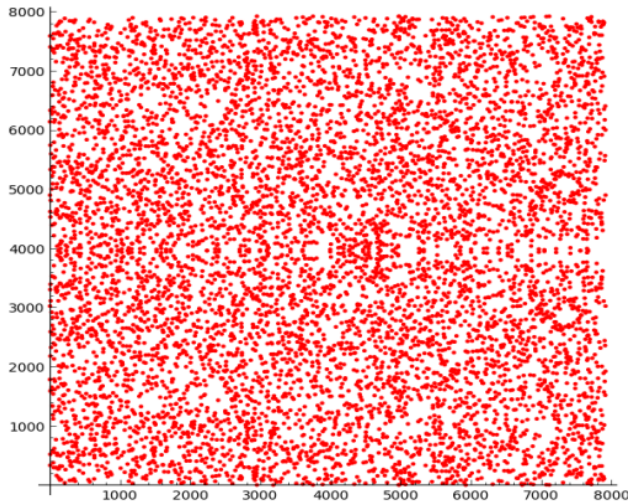
Elliptic curve implementations

- Group operation over the curve: addition and doubling



Elliptic curve implementations' details

- Above operations on the finite field:



- ...

ECDSA: Elliptic curve domain parameters

- **(field, a, b, G, n, h)**
 - Finite field
 - **p** for F_p
 - **m, bases (trinomial, pentanomial)** for F_{2^m}
 - Coefficients **a, b**: $y^2 = x^3 + ax + b$
 - Group generator: **G**
 - Order of the G: **n**
 - Optional cofactor: **h**
 - (h = number of elements in field / order n)
 - The base point G generates a cyclic subgroup of order n in the field.

ECDSA: Keys

- Generating key pair
 - Select a random integer d from $[1, n - 1]$
 - Compute $P = d * G$;
- Private key: d
- Public key: P

- For 256-bit curve
 - the private key d will be approx. 256-bit long
 - the public key P is a point on the curve – will be approx 512-bit long, unless compressed

ECDSA: Signatures

- Generate signature
 - Select a random integer k from $[1, n - 1]$
 - $(x_1, y_1) = k * G$ ←
 - Calculate $r = x_1 \pmod n$
 - Calculate $s = k^{-1}(M + r * d) \pmod n$
 - Signature is (r, s) .
- Signature verification
 - Calculate $w = s^{-1} \pmod n$
 - Calculate $u_1 = z * w \pmod n$ & $u_2 = r * w \pmod n$
 - Calculate $(x_1, y_1) = u_1 * G + u_2 * P$
 - The signature is valid if $r = x_1 \pmod n$.
- For 256-bit curve the signature length will be approx. 512 bits

ECDSA: Padding

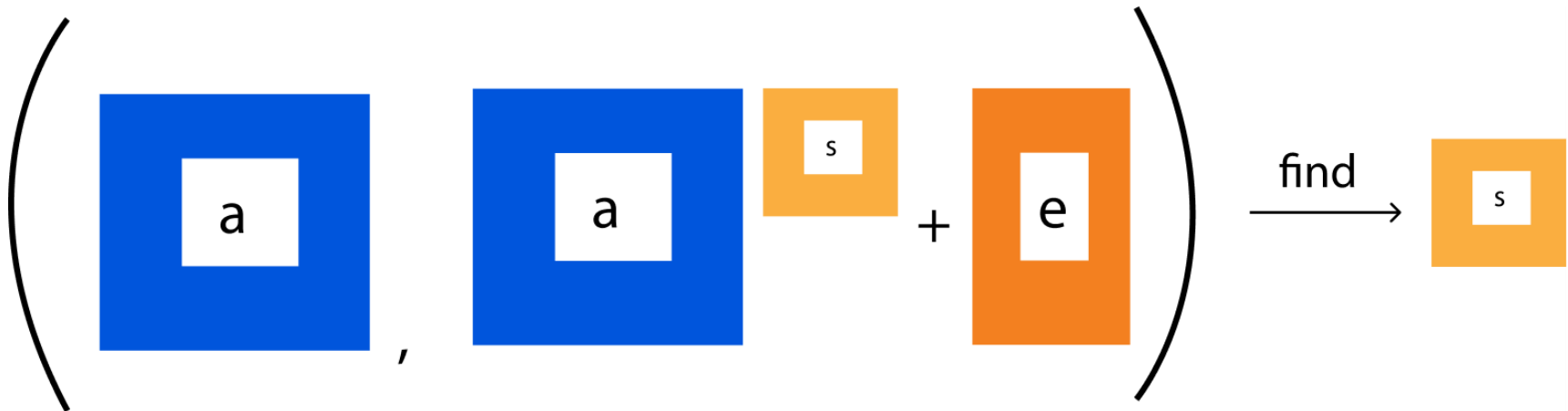
- Rules are same as for DSA
- “It is recommended that the security strength associated with the bit length of n and the security strength of the hash function be the same unless an agreement has been made between participating entities to use a stronger hash function. When the length of the output of the hash function is greater than the bit length of n , then the leftmost n bits of the hash function output block shall be used in any calculation using the hash function output during the generation or verification of a digital signature. A hash function that provides a lower security strength than the security strength associated with the bit length of n ordinarily should not be used, since this would reduce the security strength of the digital signature process to a level no greater than that provided by the hash function.” [FIPS 186-3]

Post-Quantum (PQ) Schemes

- Quantum computer breaks (polynomial time) classic public key cryptography: RSA, DSA, ECDSA
 - What about symmetric cryptography?
- NIST runs a standardization for PQ schemes:
 - <https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4>
- PQ schemes aim at higher security but are less efficient.
- Many schemes (but not all!) are based on LWE.
- Library:
 - <https://libpqcrypto.org/index.html>

The LWE problem: search and decision

- The Learning With Errors (LWE) problem asks to recover a secret vector $s=(s_1,\dots,s_n)$, where each s_i is in Z_q , given a sequence of random, “approximate” linear equations on s .



Determine if (a, b) is of the form above or random

Seminar Tasks

- A short demo based on MQDSS
 - <https://mqdss.org/>
- I used the following library:
 - <https://libpqcrypto.org/python.html>
- Signature scheme based on multivariate quadratic problem

Seminar Tasks

- First task – python console
- Files:
 - `demo.py`
 - `RSA.py` (from github)
- Good luck! 😊

Task Zero

- Let's warm up...
- I know that people called OpenSSL command line tool from python instead of using the hazmat library.
- What is happening if you use just python and hazmat?
- Open your python and run:

```
from cryptography.hazmat import backends  
backends.default_backend()
```
- How do you interpret the result?
- How many different backends there are?
- Which library is really called?

First Task – documentation:

Backend interfaces

Backend implementations may provide a number of interfaces to support operations such as [Symmetric encryption](#), [Message digests \(Hashing\)](#), and [Hash-based message authentication codes \(HMAC\)](#).

A specific `backend` may provide one or more of these interfaces.

```
class cryptography.hazmat.backends.interfaces.CipherBackend \[source\]
```

A backend that provides methods for using ciphers for encryption and decryption.

The following backends implement this interface:

- [OpenSSL backend](#)

```
cipher_supported(cipher, mode) \[source\]
```

We test compiling with `clang` as well as `gcc` and use the following [OpenSSL](#) releases:

- [OpenSSL 1.1.1-latest](#)
- [OpenSSL 3.0-latest](#)

Signature Schemes / Efficiency

- Do the tasks in `demo.py`
 - Test signature schemes
 - Measure the efficiency of some signature schemes
 - Load OpenSSL key
- Run `RSA.py` and analyze the results
 - What can you say about classical RSA vs RSA-CRT?
 - What can you say about `RSA.py` vs the python library?

Assignment 7 – Efficiency of Signature Generation Algorithms

- This is a programming assignment. Please upload your scripts/code and the required analysis via the course webpage.
- The deadline for submission is Nov. 29, 2023, 8:00.
 - -3 points for each started 24h after the deadline.
- Your code should be contained in one .py file. Please name the submission file as <uco_number>_hw7.zip. Put there both the python code, the analysis document, and all data produced during analysis (as long as the size is reasonable).
- The code must contain comments so that it is reasonably easy to understand how to run the script for evaluating each answer.

Assignment 7 - Tasks

1. Using hazmat implement the following signature schemes: RSA, DSA, ECDSA. In particular, implement key generation, signature generation, and signature verification. **[2 points]**
2. Implement HMAC with SHA2 and SHA3. All implementations should be in one file. **[1 point]**
3. Implement all the above schemes for taking input from a file (e.g., attached alice.txt). Make sure all your code works for large files (larger than your RAM). **[2 points]**
4. Perform an efficiency comparison analysis for RSA, DSA, ECDSA, HMAC-SHA2, and HMAC-SHA3. For the signature schemes' parameters, see the following slide. Analyze key generation, signature generation, and signature verification separately. Write a summary of your results, which primitive seems to be the best, and for which use case. Attach such a summary to your exercise submission. To have reliable results, perform operations a number of times and average results. **[4 points]**
Remarks: (1) For the sake of computational time, use a small message (e.g., a text file) to be signed. The same message should be used for all comparisons. For the comparison, also use the same hash function (once SHA2 and once SHA3). (2) For HMAC use the key size suggested by the symmetric crypto column in the next slide. (3) For DSA just specify the larger prime number (p). The smaller prime number (q) will be chosen automatically. (4) By "use case", I mean a scheme, for example, some algorithms are slow but have fast verification, which might make them suitable for some applications. 5) Include file reading in signature generation time.
5. Additionally, analyze whether varying the private key affects the algorithms' execution time (or the corresponding standard deviations of algorithms' execution time). Please comment on the results (what could be the reason for the observed results). **[1 point]**
6. **Bonus:** perform a similar analysis to point 4 for a larger file (approximately 100Mb) What is the difference here? What could be the reason of the change (if any)? **[1 point]**

Good luck!!!

Assignment 7 – extra info

Date	Minimum of Strength	Symmetric Algorithms	Asymmetric	Discrete Logarithm Key	Group	Elliptic Curve	Hash (A)	Hash (B)
2010 (Legacy)	80	2TDEA*	1024	160	1024	160	SHA-1** SHA-224 SHA-256 SHA-384 SHA-512	SHA-1 SHA-224 SHA-256 SHA-384 SHA-512
2011 - 2030	112	3TDEA	2048	224	2048	224	SHA-224 SHA-256 SHA-384 SHA-512	SHA-1 SHA-224 SHA-256 SHA-384 SHA-512
> 2030	128	AES-128	3072	256	3072	256	SHA-256 SHA-384 SHA-512	SHA-1 SHA-224 SHA-256 SHA-384 SHA-512
>> 2030	192	AES-192	7680	384	7680	384	SHA-384 SHA-512	SHA-224 SHA-256 SHA-384 SHA-512
>>> 2030	256	AES-256	15360	512	15360	512	SHA-512	SHA-256 SHA-384 SHA-512

