Privacy-Preserving Systems
(a.k.a., Private Systems)

CU Graduate Seminar

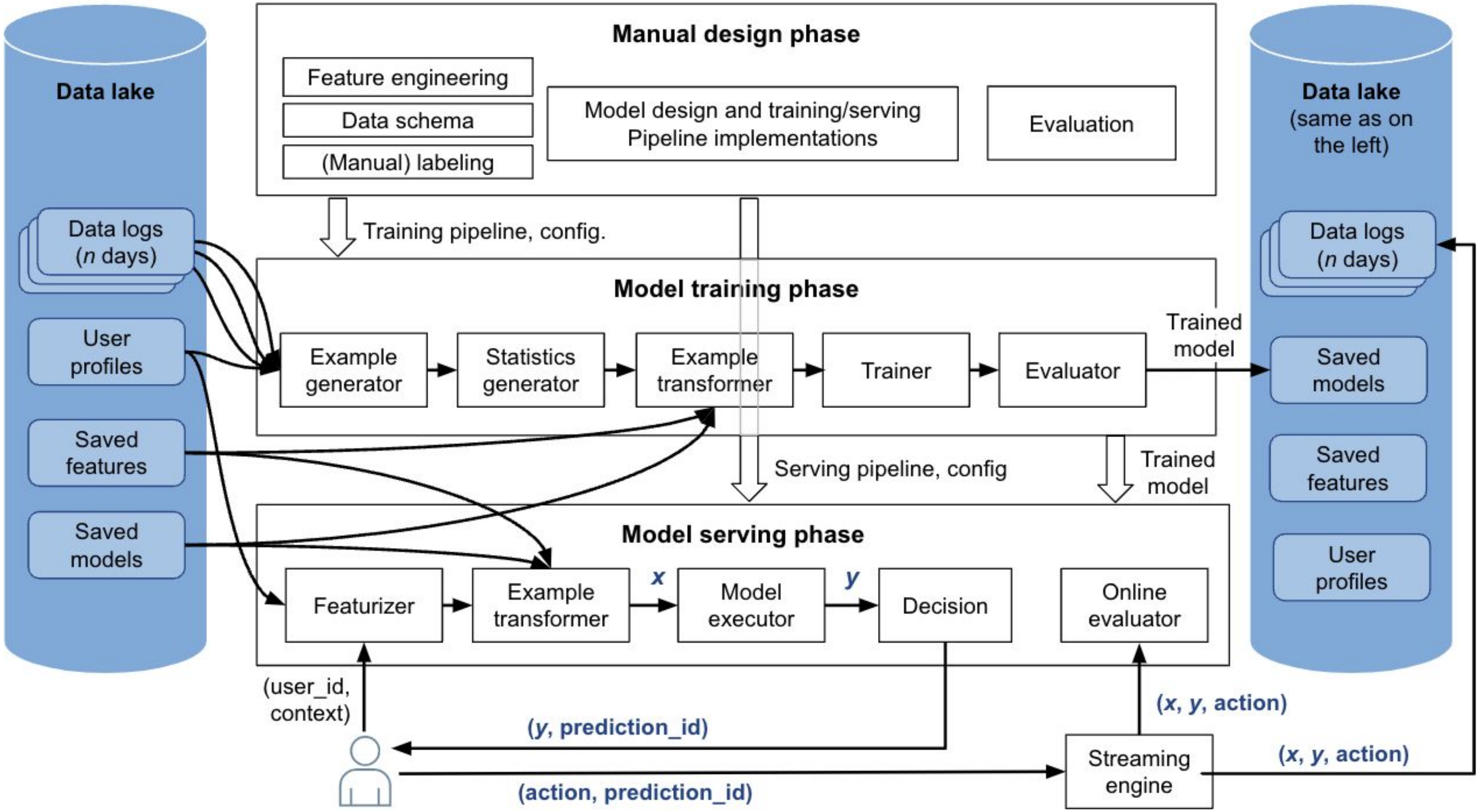
Instructor: Roxana Geambasu

Homomorphic Encryption

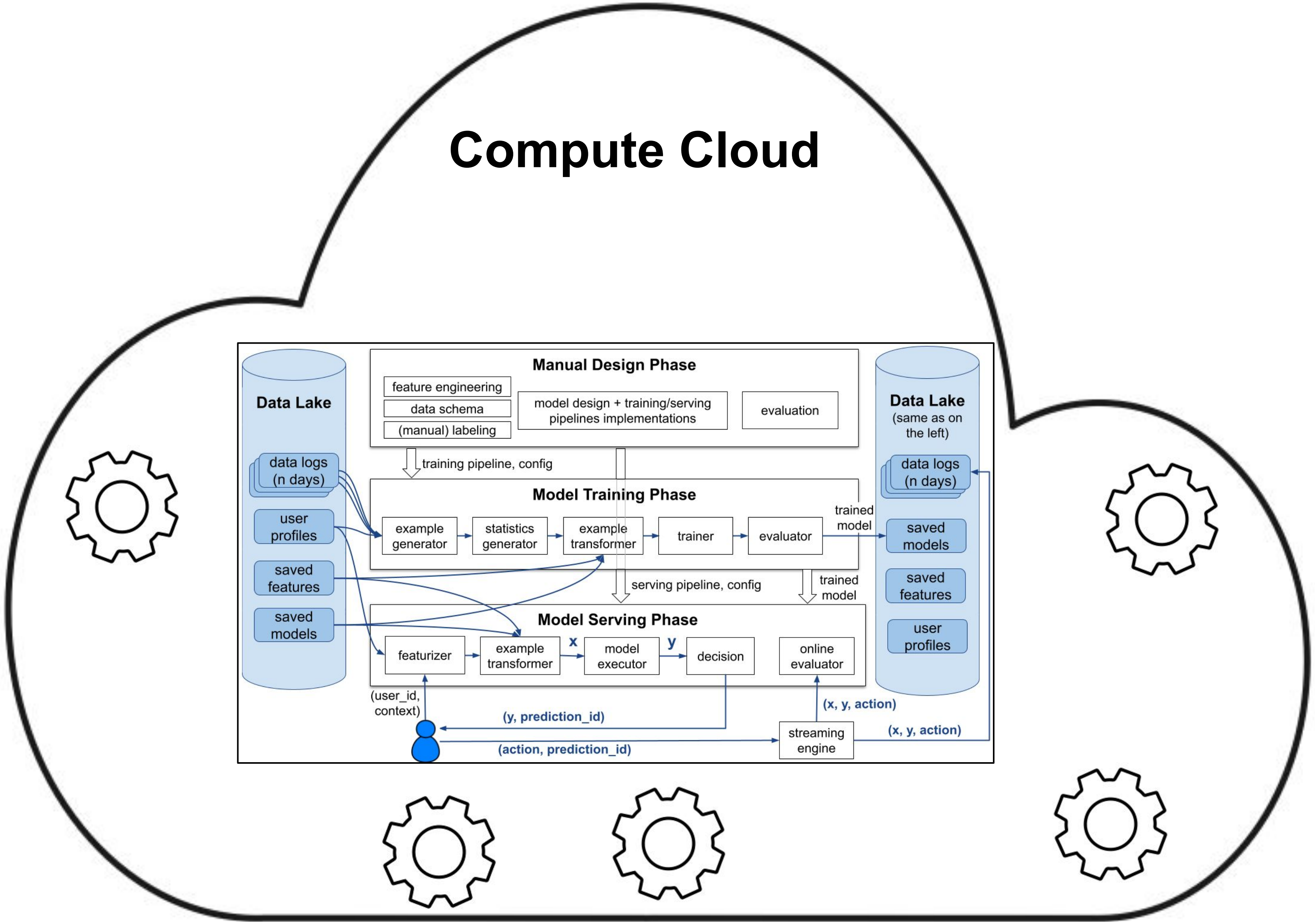
Acknowledgement: This lecture was inspired by this 2019 talk by Prof. Raluca Ada Popa

Limitations of Traditional Encryption for Data Exposure Risks in (ML) Clouds

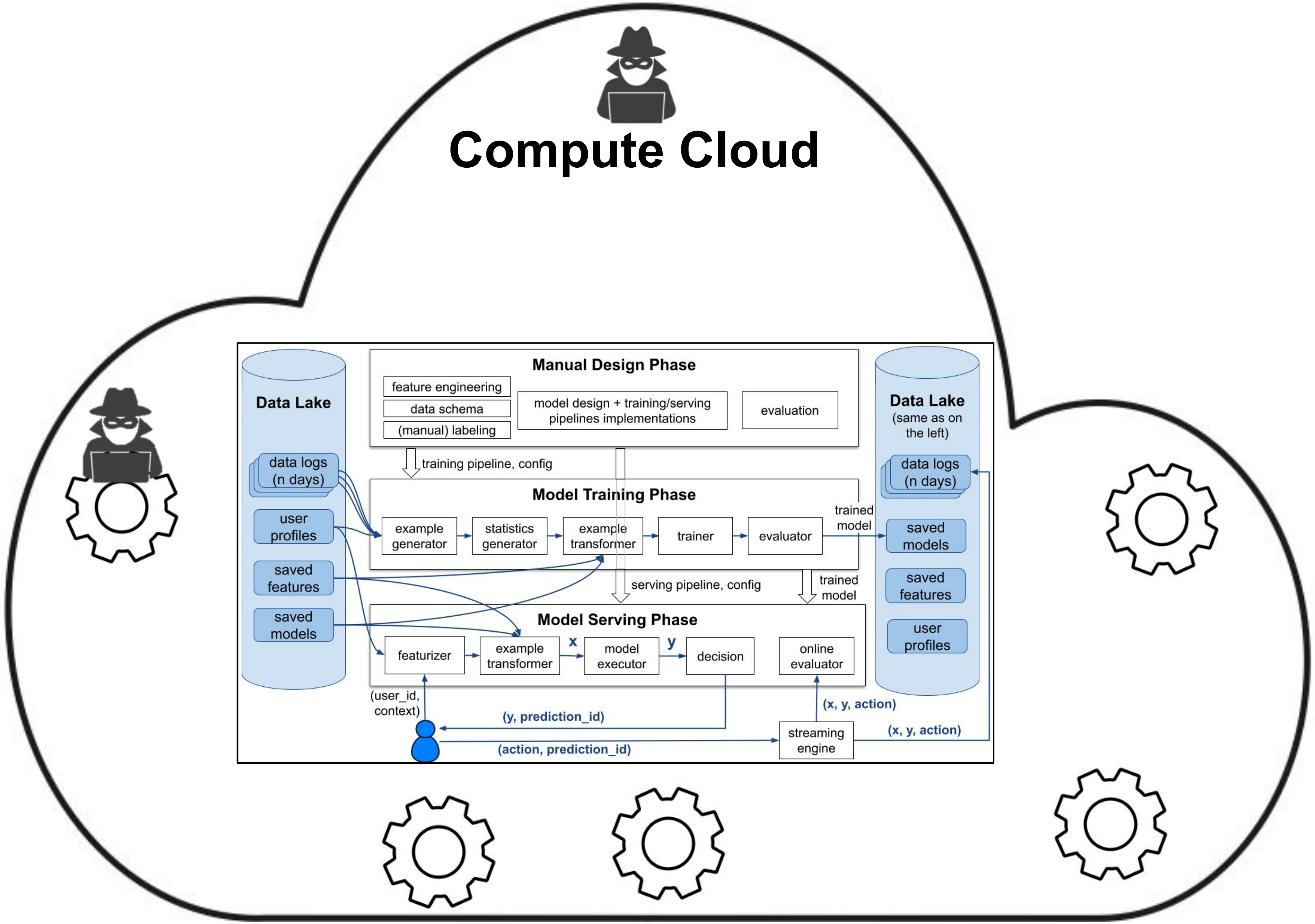
Reminder: Data Risks in ML Ecosystems



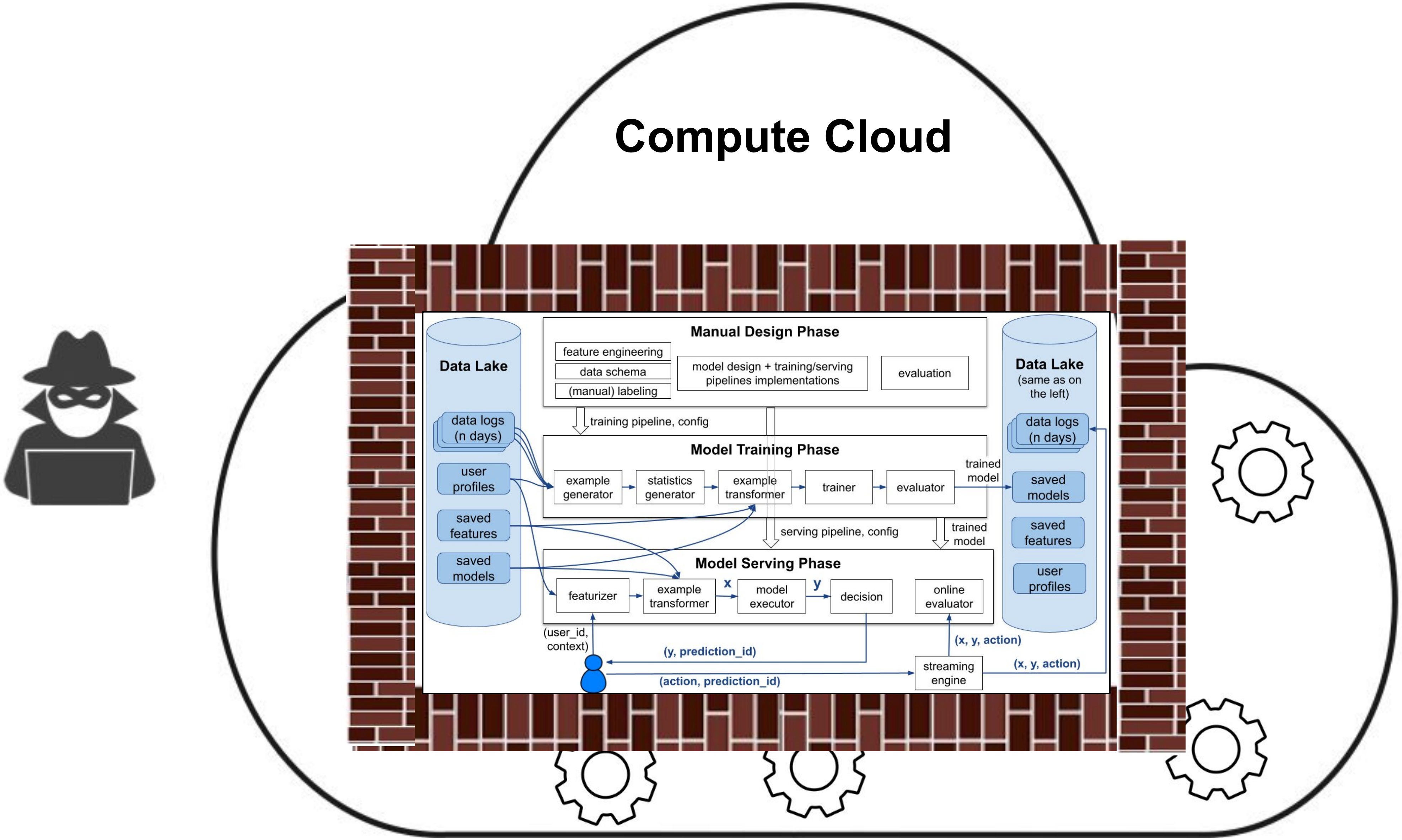
Clouds Add Further Risks



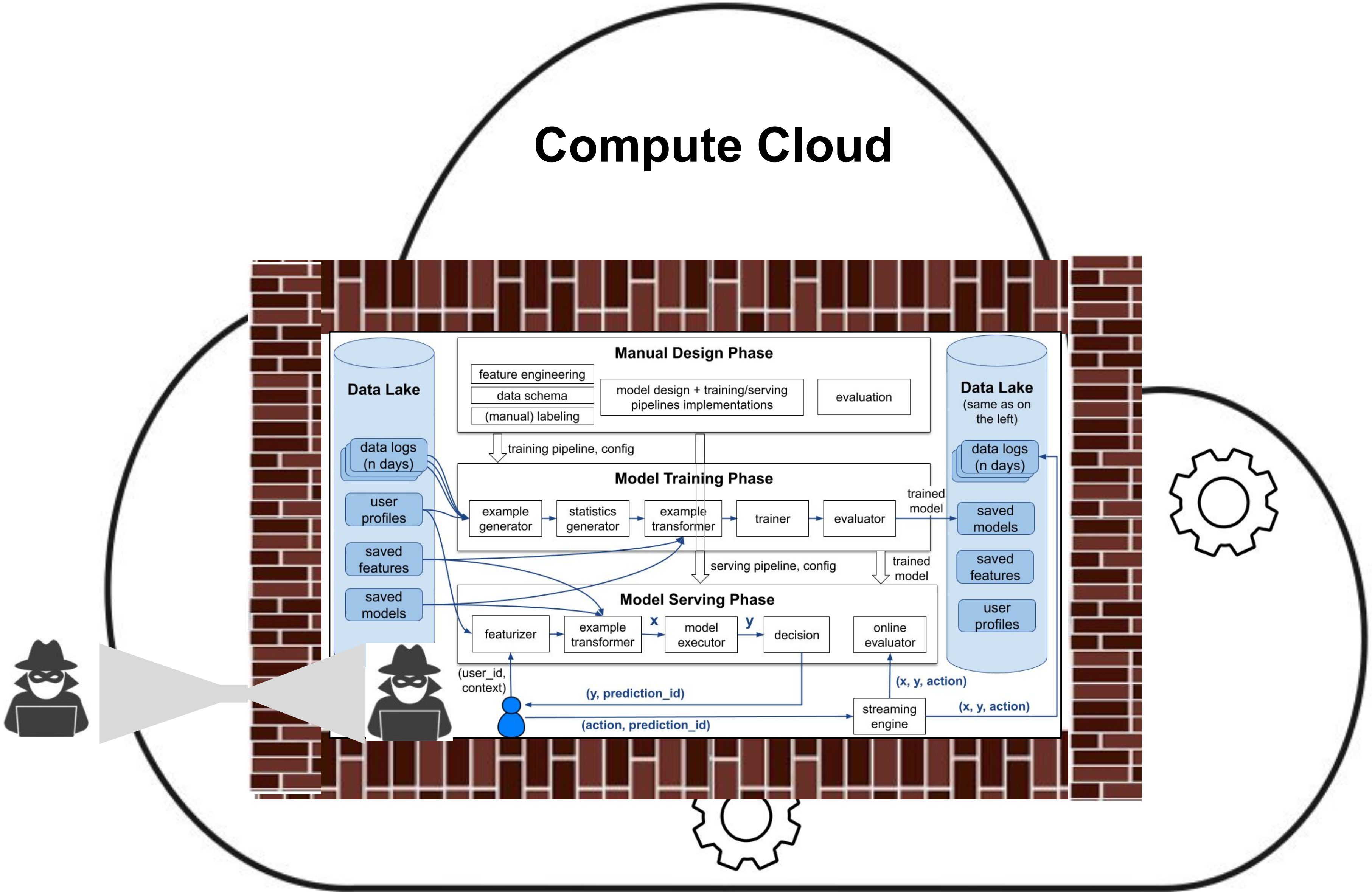
Clouds Add Further Risks



Traditional Security's Main Weakness



Attackers Eventually Break In

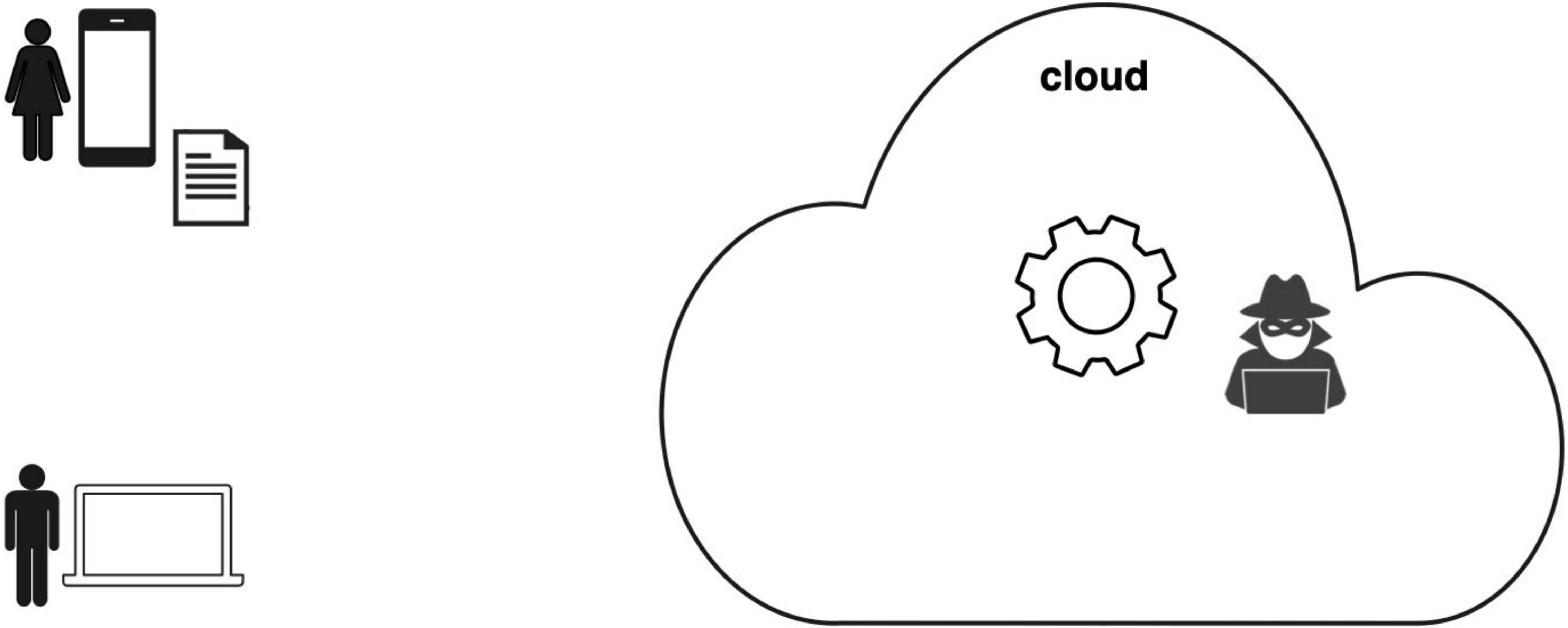


Assume the Attacker Will Break In

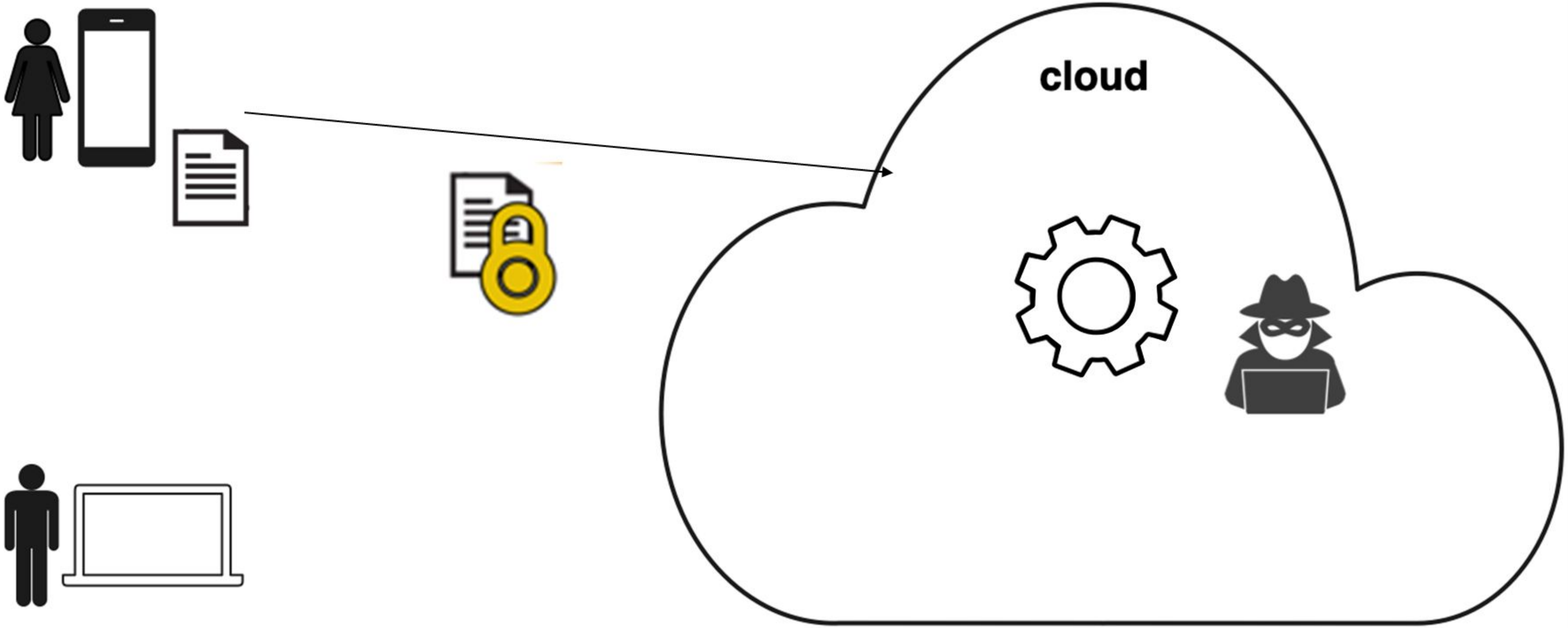
“in the cloud [...] applications need to protect themselves instead of relying on firewall-like techniques”

Werner Vogels,
Amazon CTO

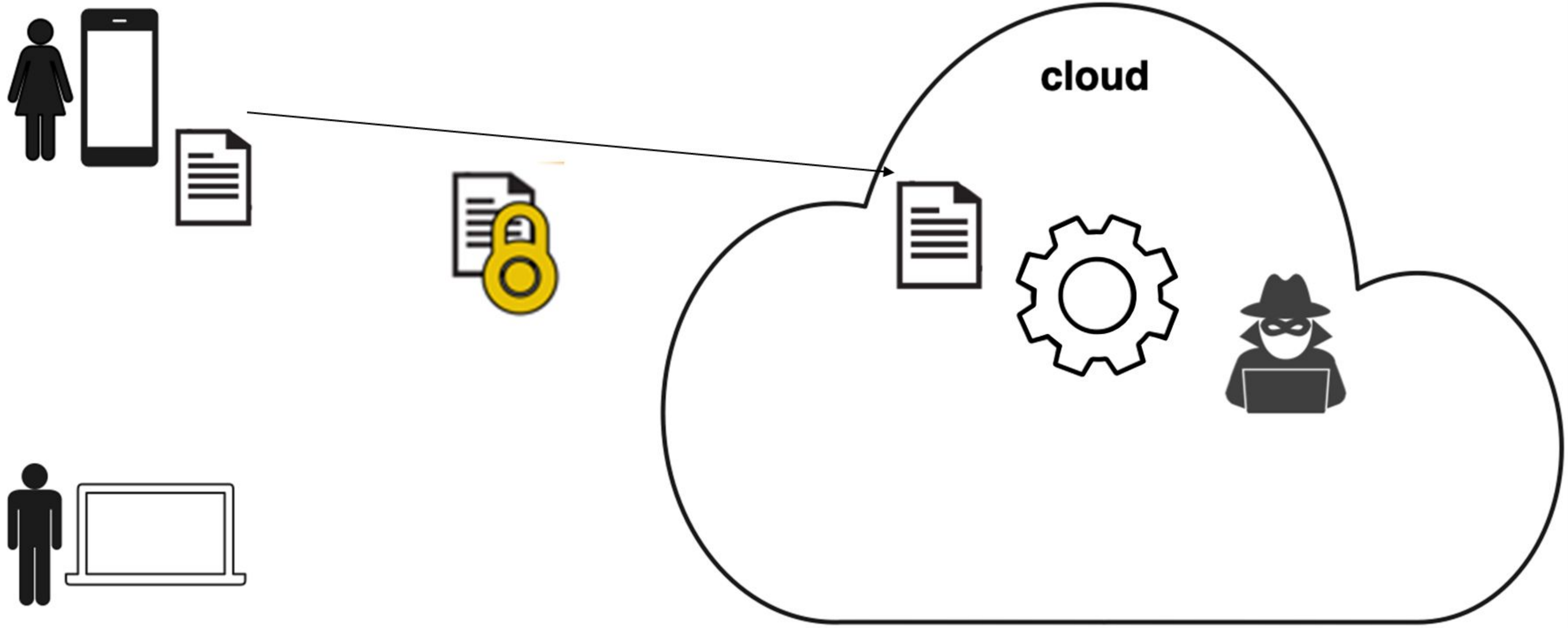
Standard Use of Encryption



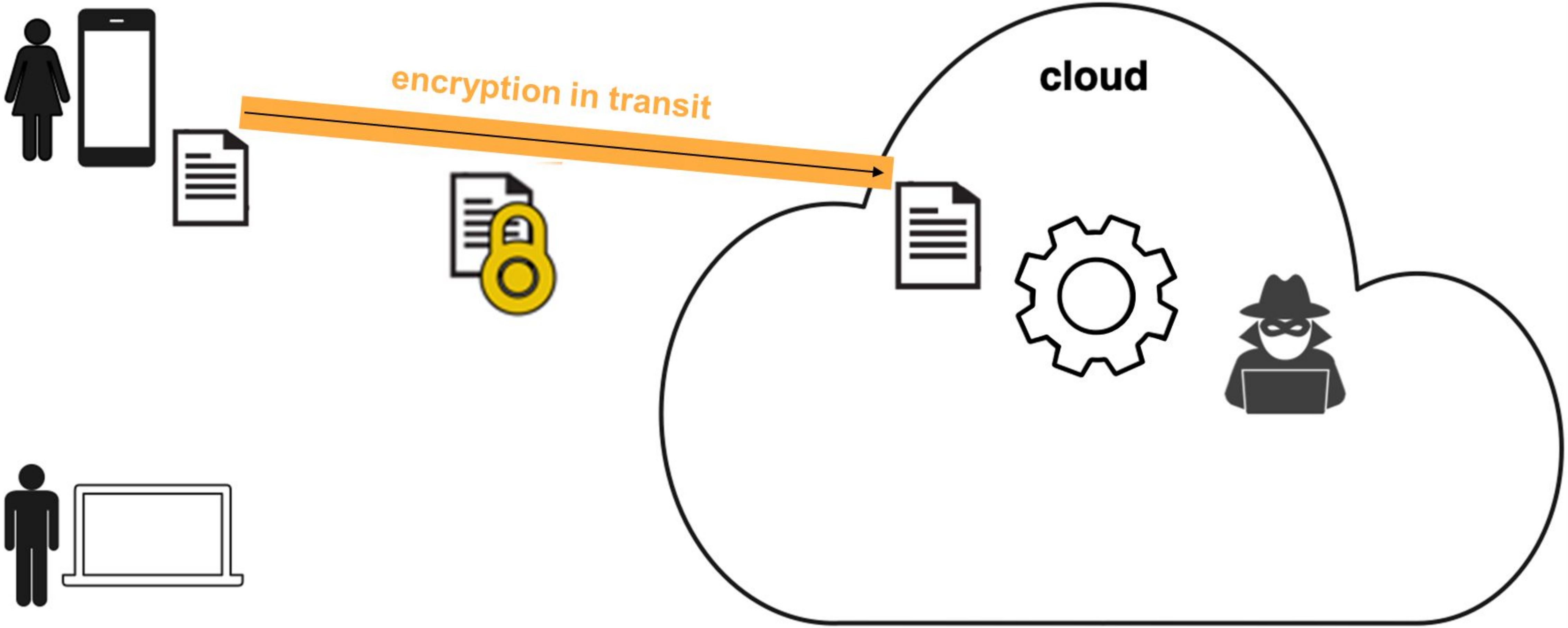
Standard Use of Encryption



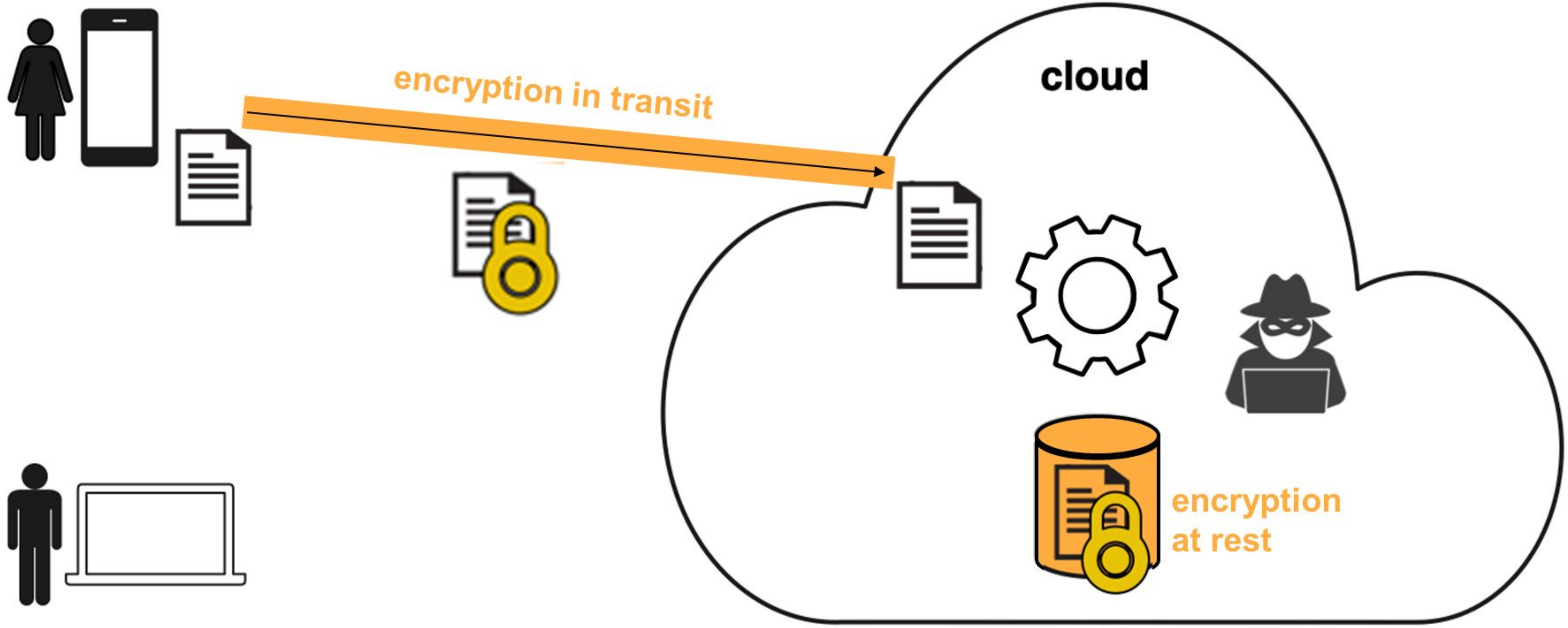
Standard Use of Encryption



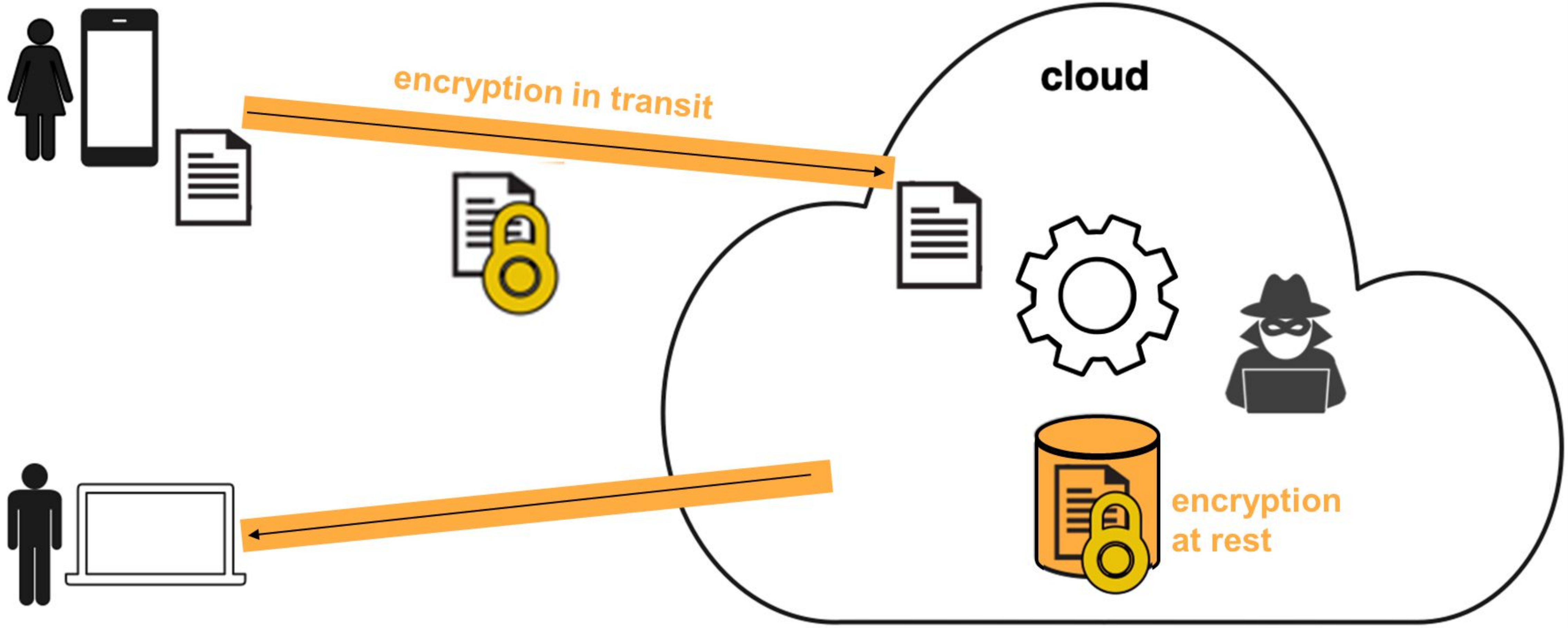
Standard Use of Encryption



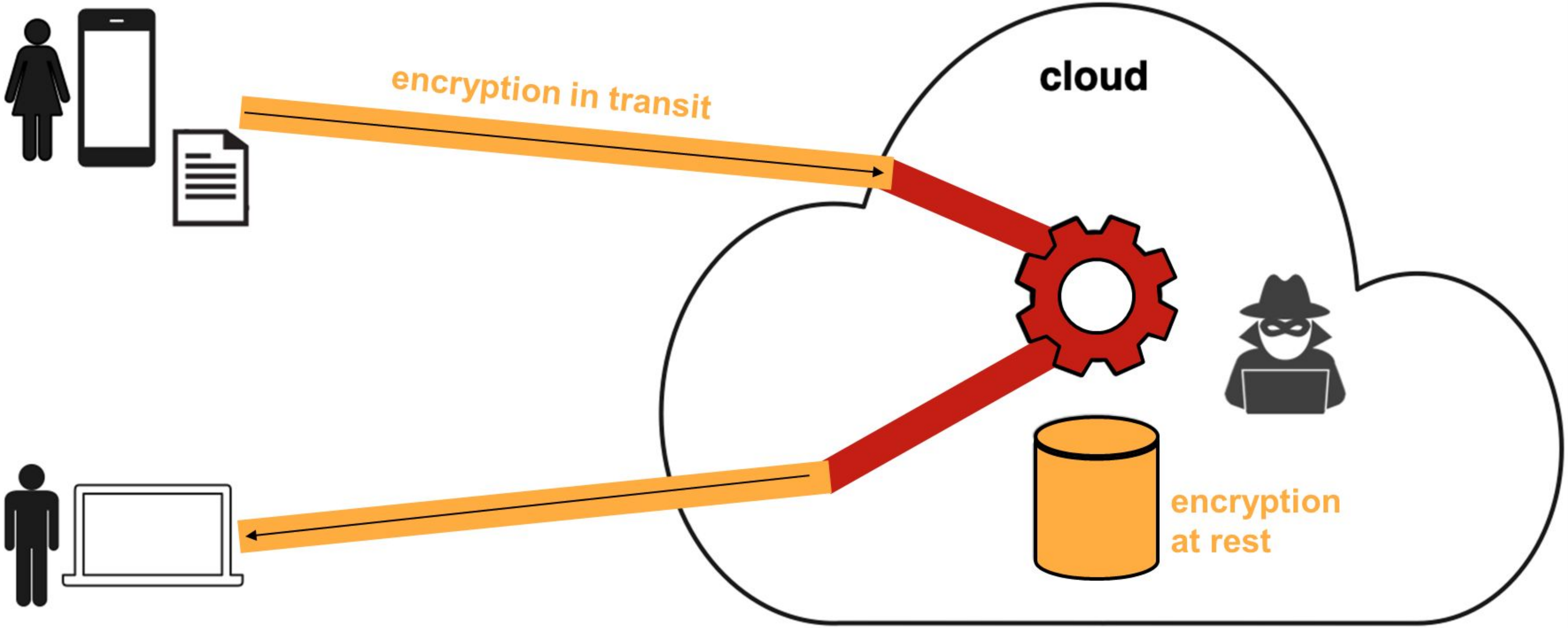
Standard Use of Encryption



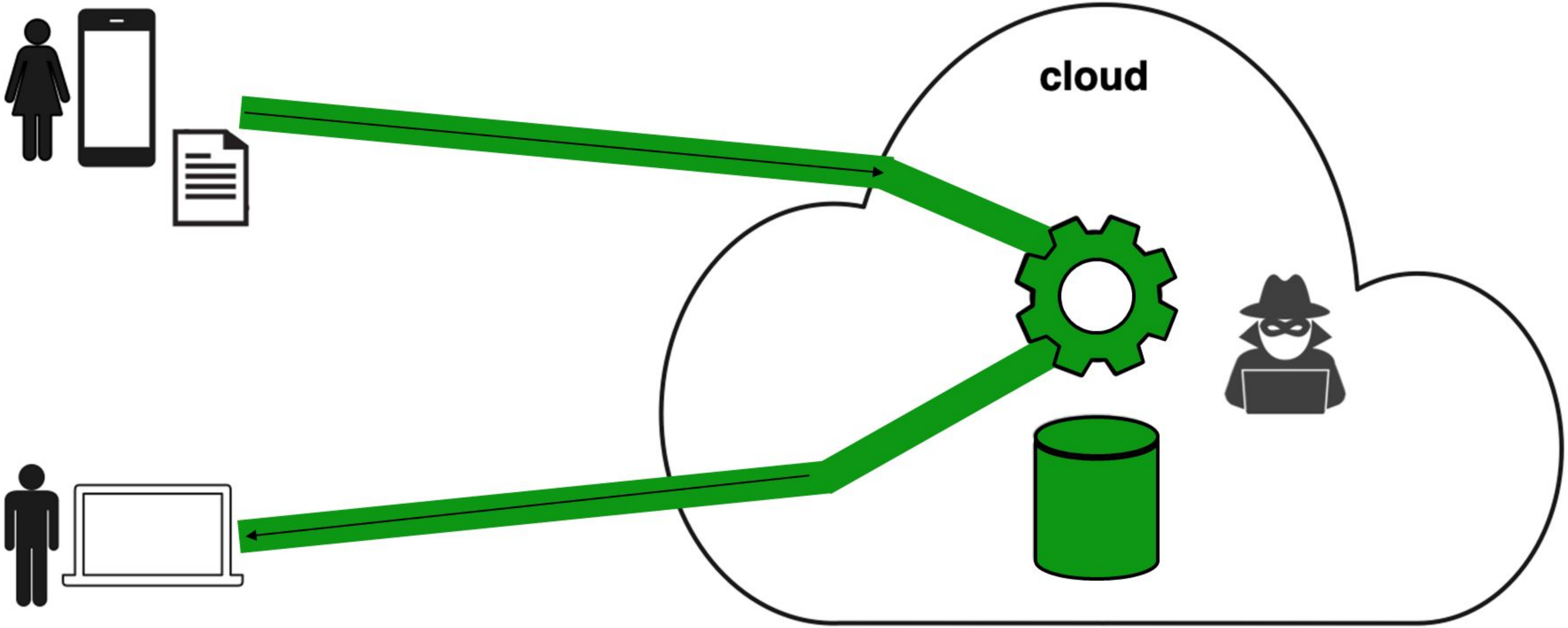
Standard Use of Encryption



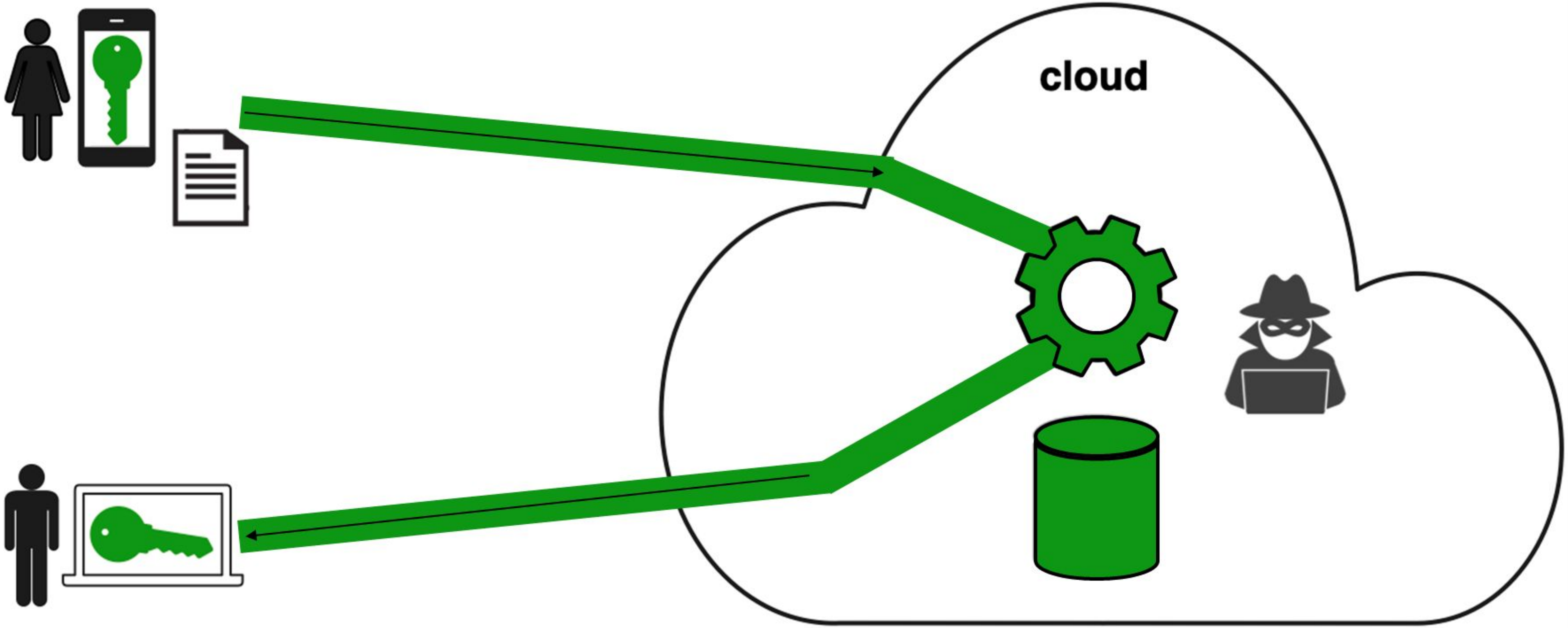
Standard Use of Encryption



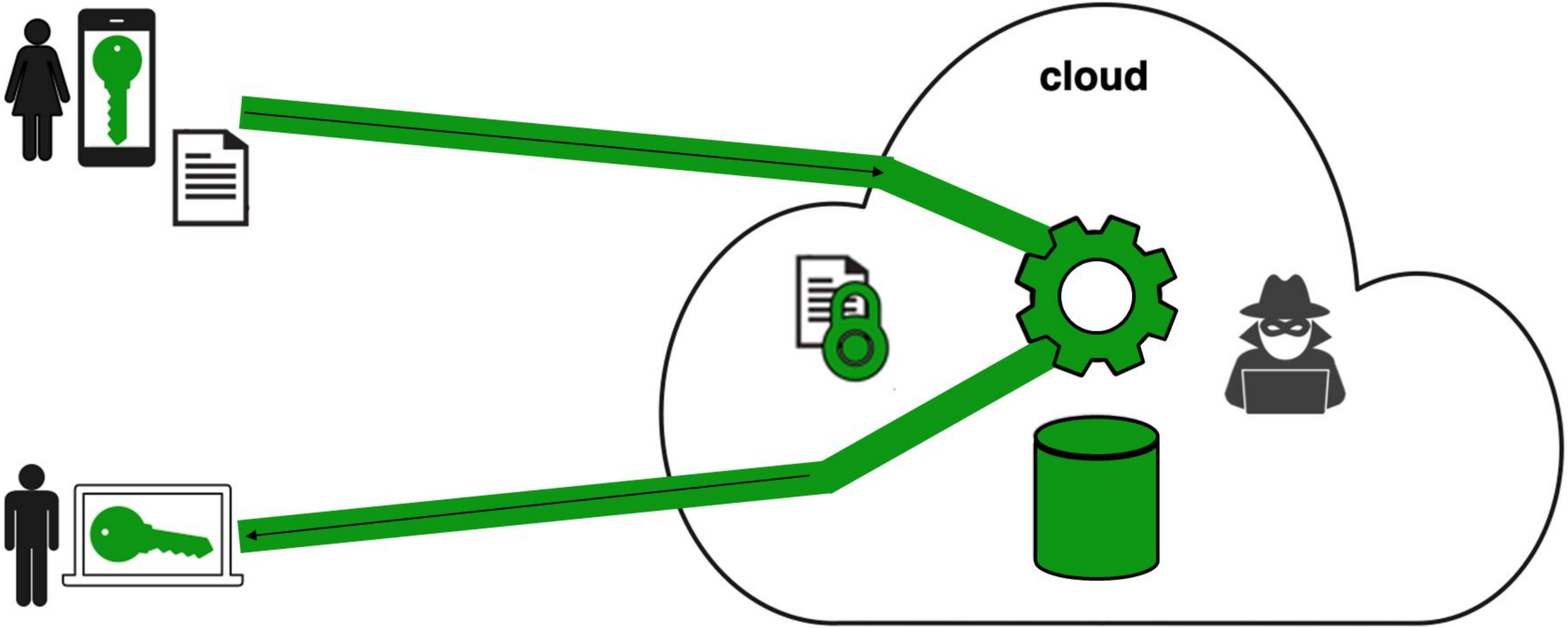
Need: Encryption With Computation



Need: Encryption With Computation



Need: Encryption With Computation



Advanced Cryptography

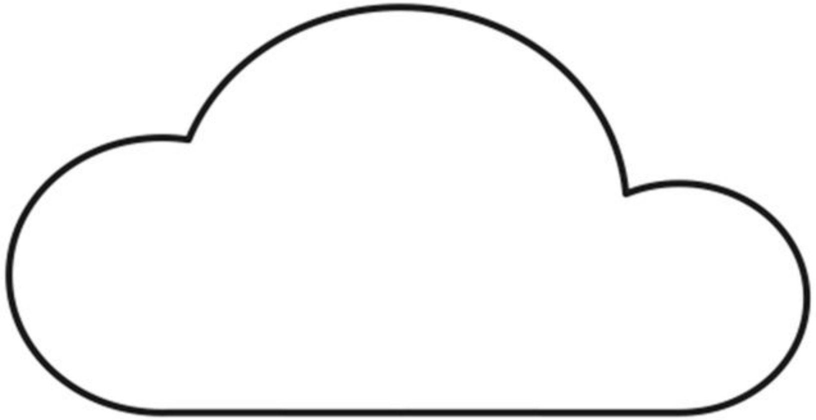
- Homomorphic encryption
- Secure multiparty computation
  - Related: federated learning
  - Together, we discuss these as “private collaborative learning”
- Secure enclaves
- Our goal: overview these so learners have a springboard for learning more

Limitations of Traditional Encryption for Data Exposure Risks in (ML) Clouds

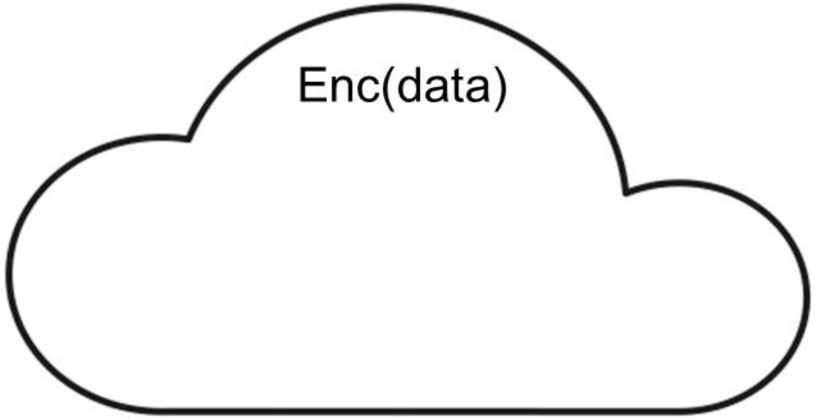
The End

Homomorphic Encryption Overview

Computation on Encrypted Data



Computation on Encrypted Data

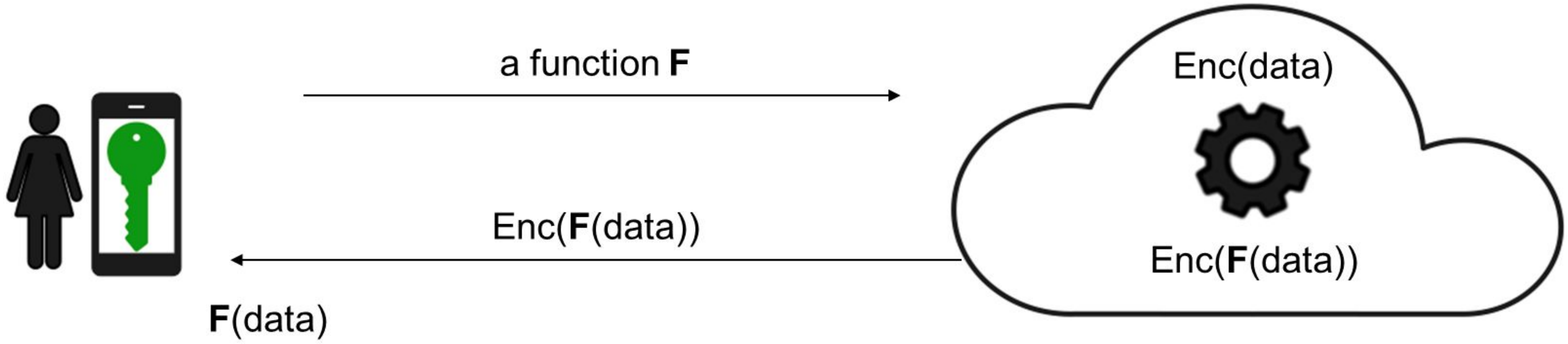


Computation on Encrypted Data

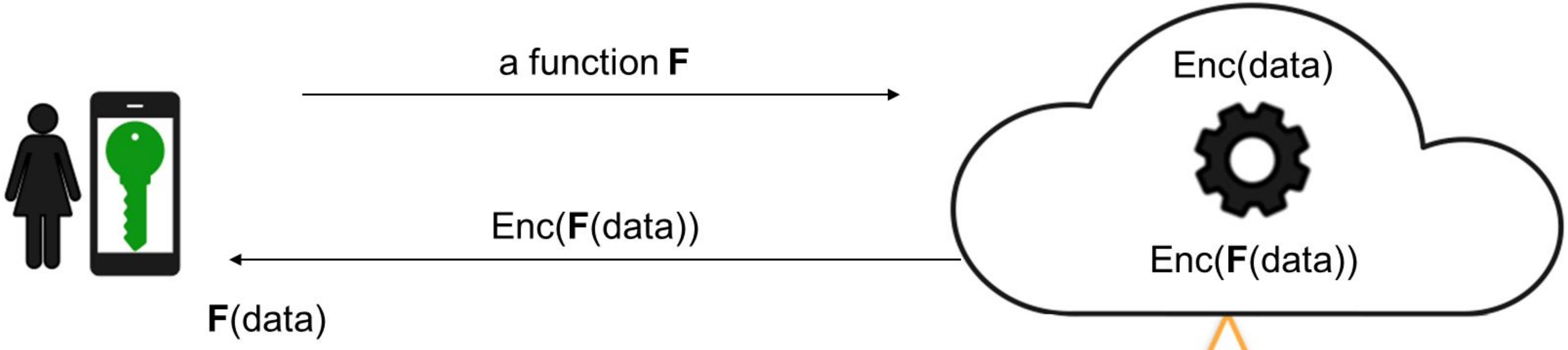
Computation on Encrypted Data

Computation on Encrypted Data

Computation on Encrypted Data



Computation on Encrypted Data

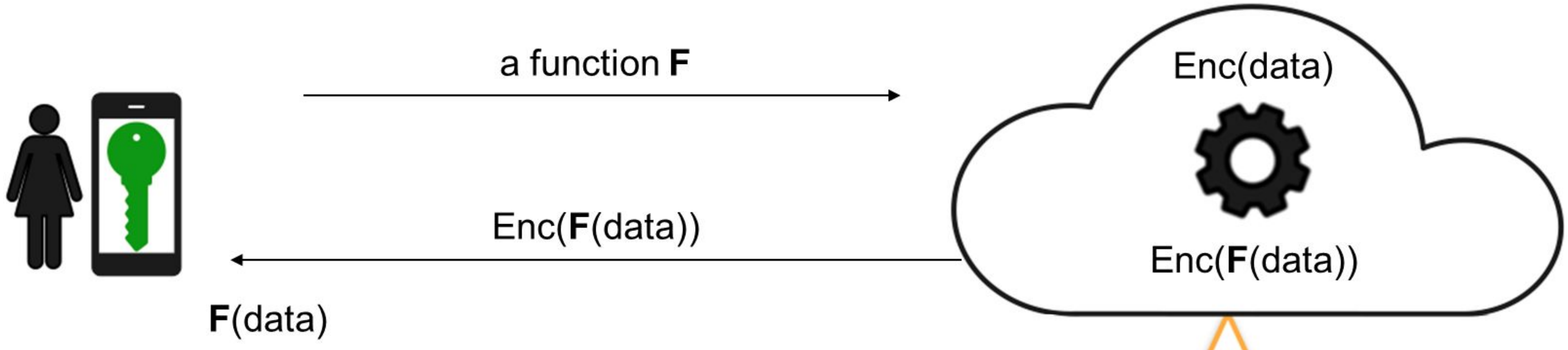


Example: RSA public key encryption, **F = ***

$\text{Enc}(x) = x^e \bmod n$
$\text{Enc}(y) = y^e \bmod n$
--- (multiply)
$\text{Enc}(x)*\text{Enc}(y) = (xy)^e \bmod n = \text{Enc}(x*y)$

i.e., RSA is multiplicatively homomorphic

Computation on Encrypted Data



Example: Paillier cryptosystem, **F** = +

$\mathrm{Enc}(x) = g^x r^n \bmod n^2$

$\mathrm{Enc}(y) = g^y r^n \bmod n^2$

(multiply)

$\mathrm{Enc}(x) * \mathrm{Enc}(y) = g^{x+y}(rr')^n \bmod n^2 = \mathrm{Enc}(x+y)$

i.e., Paillier is additively homomorphic

Fully Homomorphic Encryption

- Enables **general functions** on encrypted data
- Despite progress, remains orders of magnitude too slow.
- However, specialized homomorphic encryption schemes, developed for specific operations, are practical.
- Numerous useful systems have been developed, which are worth considering to deploy in one's most vulnerable/exposed components.

[Gentry09]

Homomorphic Encryption Overview

The End

Background/Math behind These Schemes

Cryptography Basics

- Goal: allow intended recipients of a message to receive the message securely:
    - Confidentiality
    - Integrity
    - Non-repudiation
- Two types:
    - Public-key or Symmetric-key
    - Public-key or Asymmetric-key

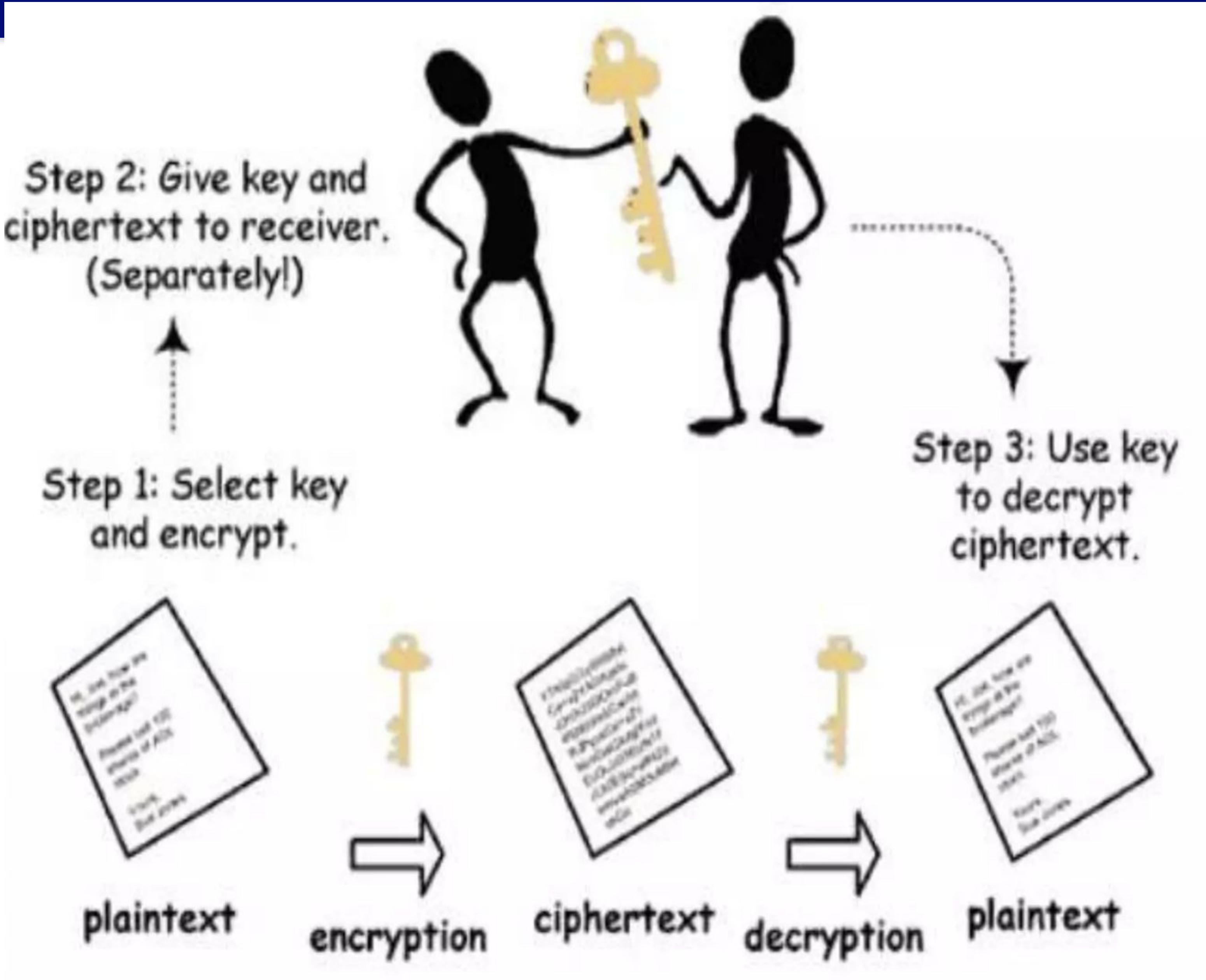
This and next few slides were inspired by this slide deck.

Important terms

- Plaintext -- the message in its original form.
- Ciphertext -- message altered to be unreadable by anyone except intended recipients.
- Cipher -- The algorithm used to encrypt the message.
- Cryptosystem -- The combination of algorithm, key, and key management functions used to perform cryptographic operations.

Private Key Cryptography

- A single key is used for both encryption and decryption. That’s why it’s called “symmetric” key as well.
- The sender uses the key to encrypt the plaintext and the receiver applies the same key to decrypt the message.
- The biggest difficulty with this approach is thus the distribution of the key, which generally a trusted third-party does.

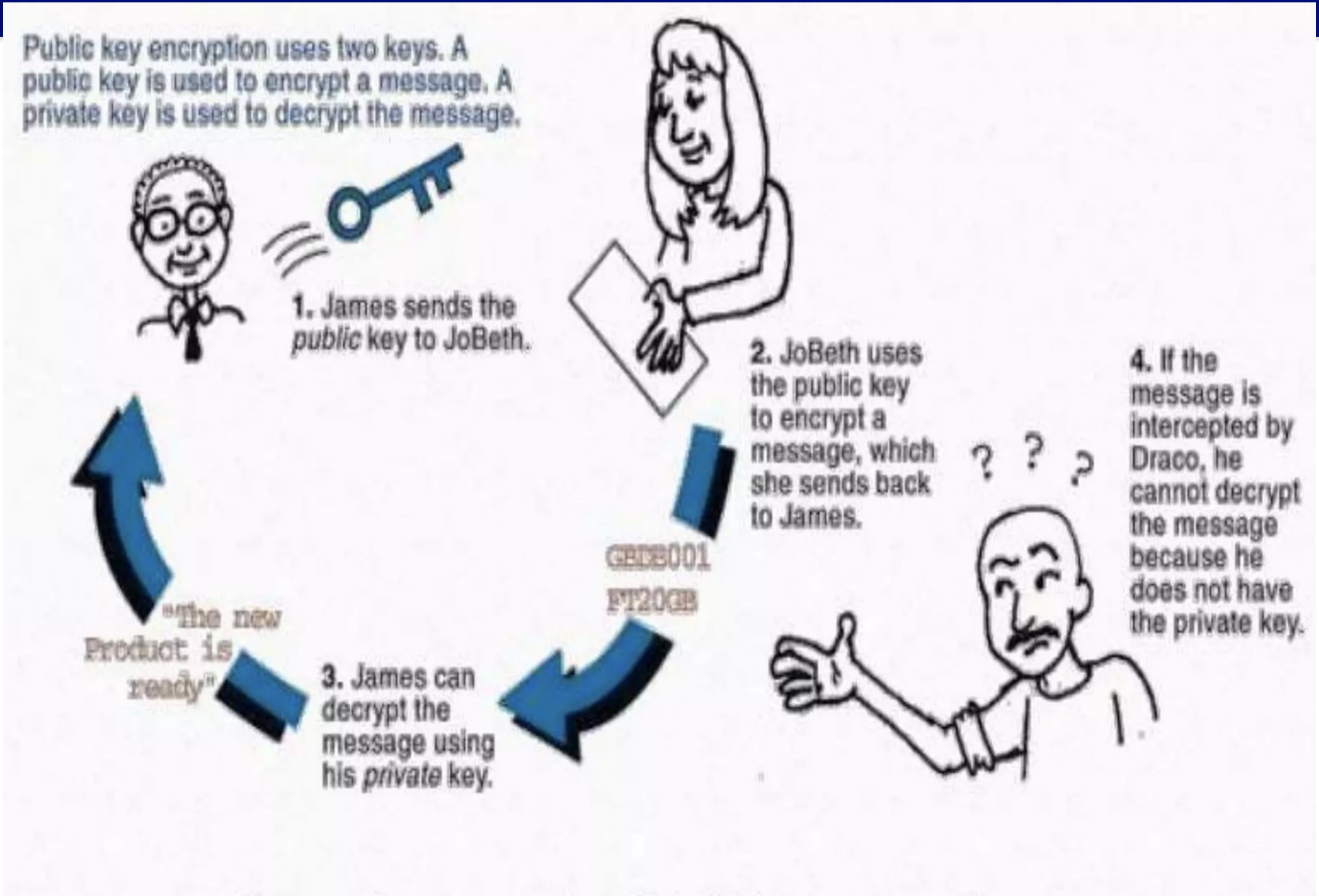


Schematic representation of Private-key cryptography

Schematic from [here](#).

Public-Key Cryptography

- Each user has a pair of keys: a public key and a private key.
- The public key is used for encryption. This is released in public (usually through PKI).
- The private key is used for decryption. This is known to the owner only.



Schematic representation of Public-key cryptography

Schematic from [here](#).

RSA Cryptosystem

- Most famous public-key algorithm used today is RSA.
  - Developed in 1976 by MIT scientists, Ronald Rivest, Adi Shamir, Leonard Adleman.
- Used in hundreds of software products and can be used for digital signatures, or encryption of small blocks of data (such as to establish symmetric session keys).
- Relies on the relative ease of finding large primes and the comparative difficulty of factoring large integers for its security.

Algorithms

Key generation
Encryption
Decryption

RSA Key Generation

$\phi(n)$ = Euler’s totient function (in this case, because n=pq and p,q primes, $\phi(n) = (p-1)(q-1)$)

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

RSA Encryption, Decryption

Encryption

- Plaintext: $M < n$
- Ciphertext: $C = M^e \pmod{n}$

Decryption

- Ciphertext: C
- Plaintext: $M = C^d \pmod{n}$

Key Generation

- Find two large primes, p and q.
- Form their product $n = pq$.
- Choose random integer e, which is relatively prime to $(p-1)(q-1)$.
- **The pair (n,e) is the public key.**
- Use Extended Euclid's Algorithm and Euler's Theorem to calculate d, which is e's modular inverse.:

$$ed \equiv 1 \pmod{(p-1)(q-1)}$$

- **The pair (n,d) is the private key.**
  - Like d, factors p,q must be kept secret (they can be destroyed after d is generated).

RSA is Multiplicatively Homomorphic

$\text{Enc}(x) = x^e \bmod n$
$\text{Enc}(y) = y^e \bmod n$
-- (multiply ciphertexts)

$\text{Enc}(x)*\text{Enc}(y) = (xy)^e \bmod n = \text{Enc}(x*y)$ (to get the ciphertext of the multiplication of the cleartexts)

RSA is not known to be additively homomorphic.

Paillier Cryptosystem

- Similar assumptions as RSA, but it is additively homomorphic.
  - And not known to be multiplicatively homomorphic…

- (Paillier is also secure against chosen-plaintext attack, which RSA on its own is not.)

Next few slides were inspired by: hhttps://www.slideshare.net/DejanRadi1/paillier-cryptosystem

Paillier Key Generation

1. Pick two large prime numbers p and q, randomly and independently. Confirm that $\gcd(pq, (p-1)(q-1))$ is 1. If not, start again. [Loop]
2. Compute $n = pq$.
3. Define function $L(x) = \frac{x-1}{n}$.
4. Compute λ as $\mathrm{lcm}(p-1, q-1)$.
5. Pick a random integer g in the set $\mathbb{Z}^*_{n^2}$ (integers between 1 and n^2).
6. Calculate the modular multiplicative inverse $\mu = (L(g^\lambda \mod n^2))^{-1} \mod n$. If μ does not exist, start again from step 1. [Loop]
7. The public key is (n, g). Use this for encryption.
8. The private key is λ. Use this for decryption.

Paillier Encryption, Decryption

Encryption can work for any m in the range $0 \leq m < n$:

1. Pick a random number r in the range $0 < r < n$.
2. Compute ciphertext $c = g^m \cdot r^n \mod n^2$.

Decryption presupposes a ciphertext created by the above encryption process, so that c is in the range $0 < c < n^2$:

1. Compute the plaintext $m = L(c^\lambda \mod n^2) \cdot \mu \mod n$.

(Reminder: we can always recalculate μ from λ and the public key).

Paillier is Additively Homomorphic

$$\mathrm{Enc}(x) = g^x r^n \bmod n^2$$

$$\mathrm{Enc}(y) = g^y r^n \bmod n^2$$

(multiply the ciphertexts)

$$\mathrm{Enc}(x) * \mathrm{Enc}(y) = g^{x+y}(rr')^n \bmod n^2 = \mathrm{Enc}(x+y)$$

(to get the ciphertext of the addition of the cleartexts)

Paillier is not known to be multiplicatively homomorphic.

AES Cryptosystem

- Symmetric-key system
- Used to encrypt messages once a session has been established.
- Much faster than public-key encryption!
- Doesn’t rely on difficult number-theory problem, but rather on passing the cleartext through many transformation blocks that no one knows how to break (yet?).
- Is not homomorphic, but its “deterministic” mode, which is vulnerable to chosen-plaintext attacks, can support equality comparisons, hence it is sometimes used in encrypted computation systems (b/c it’s a cheap alternative to other deterministic encryption schemes).
  - (and you will use it in HW3)

Next few slides inspired from this slide deck

How AES Works

- Repeats 4 main functions to encrypt data.
- Takes 128-bit block of data and a key and gives ciphertext as output.
- Functions are:

    I. Sub Bytes
    II. Shift Rows
    III. Mix Columns
    IV. Add Key

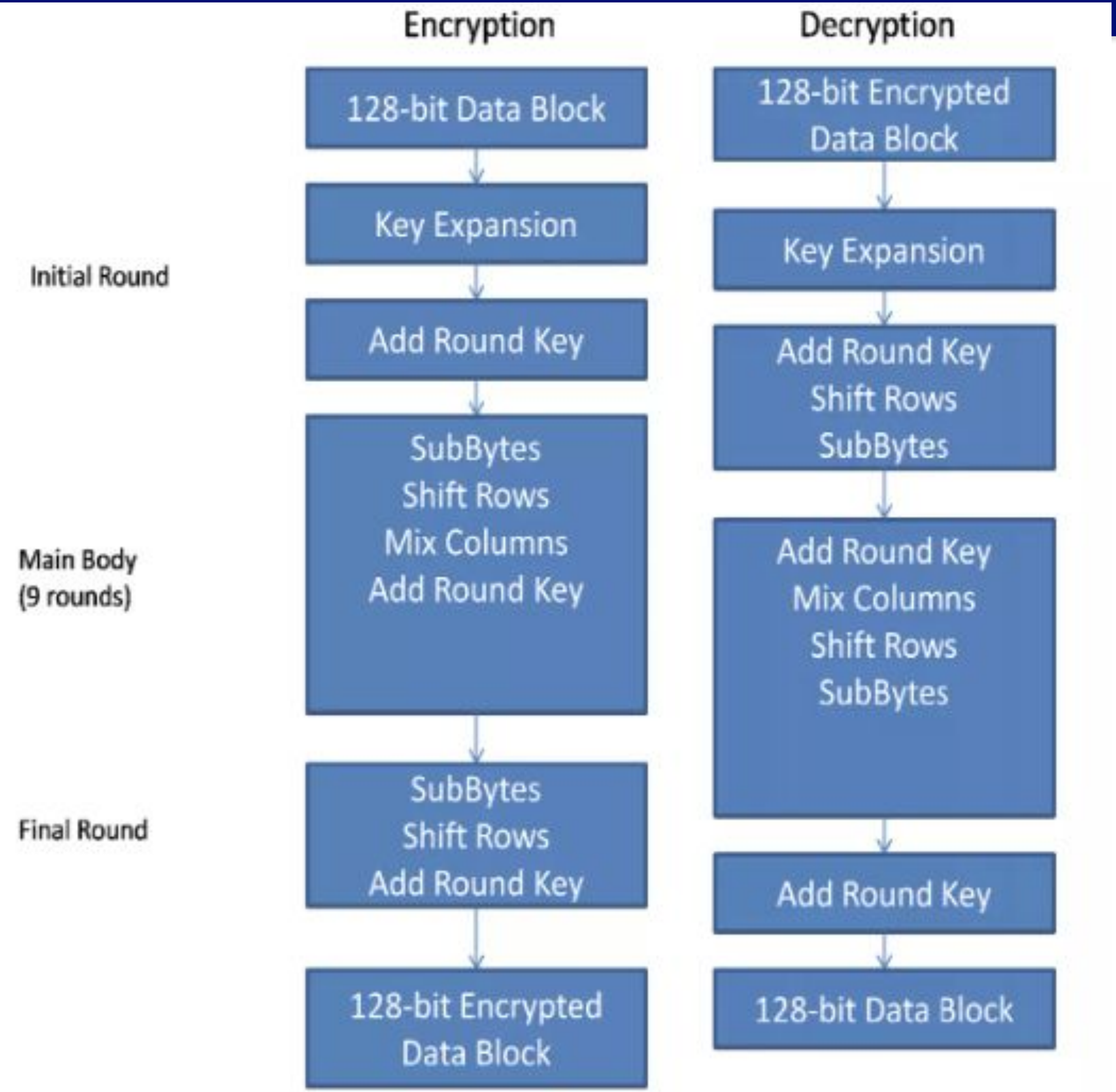
How AES Works (cont.)

- The number of rounds performed by the algo depends on the key size.

Key size (bits)	Rounds
128	10
192	12
256	14

- Tradeoff between security and runtime (but in any case, much faster and memory efficient than RSA for example).

Schematic of AES block cipher



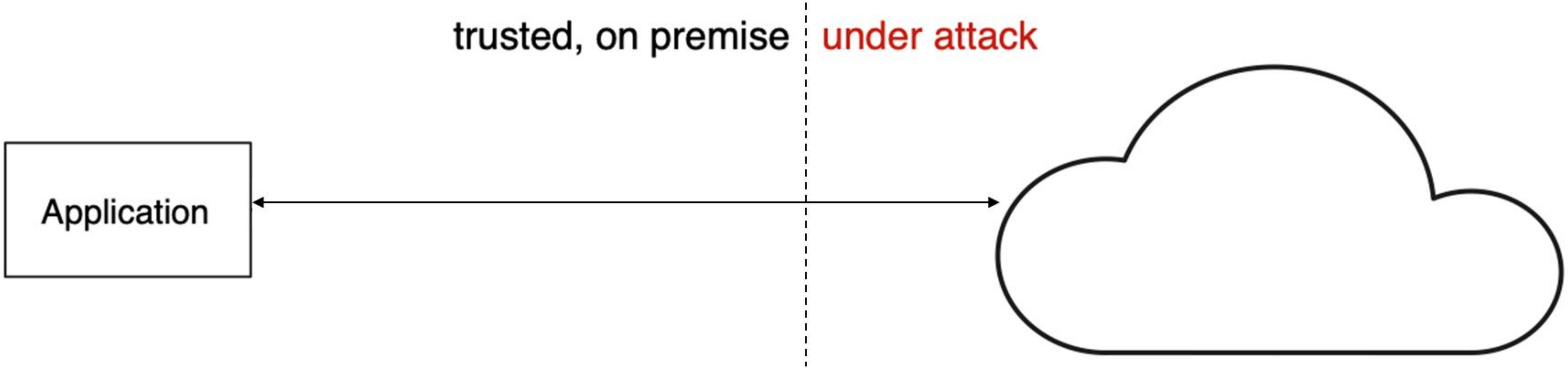
Background/Math behind These Schemes

The End

Example System: Encrypted Database

Encrypted Databases

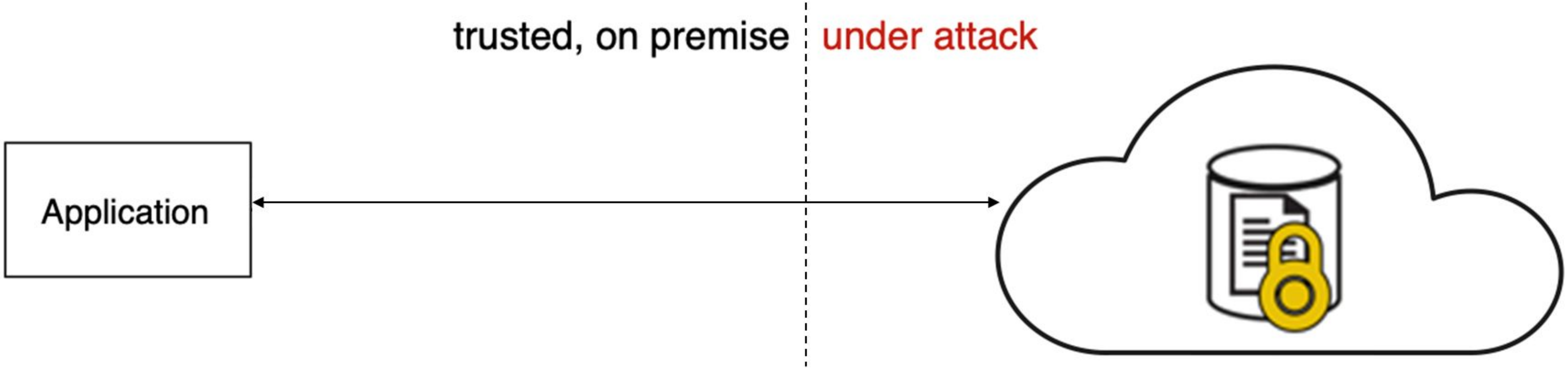
CryptDB (Popa11) was a first DBMS to process SQL queries on encrypted data.



[Popa11]

Encrypted Databases

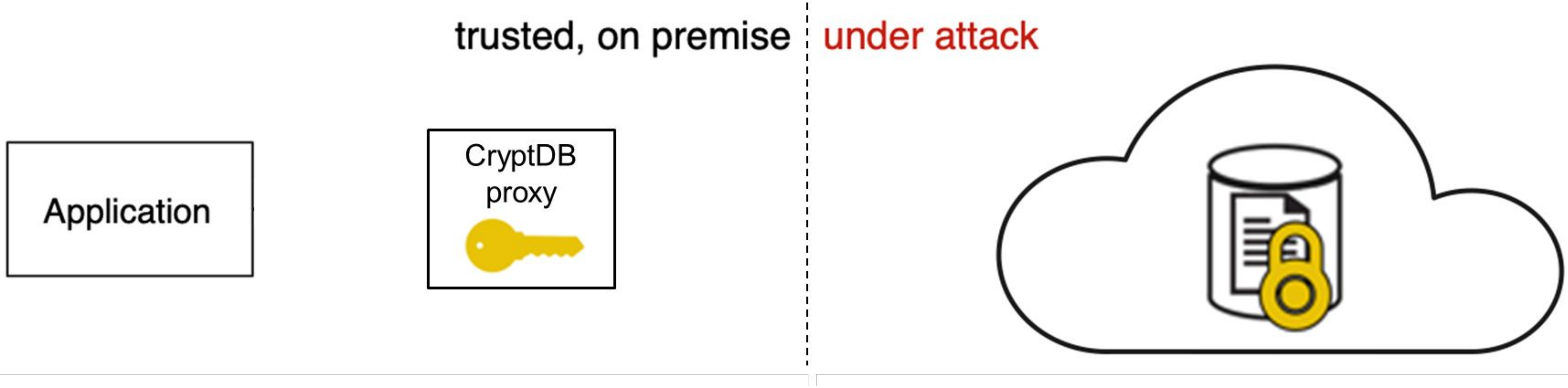
CryptDB (Popa11) was a first DBMS to process SQL queries on encrypted data.



[Popa11]

Encrypted Databases

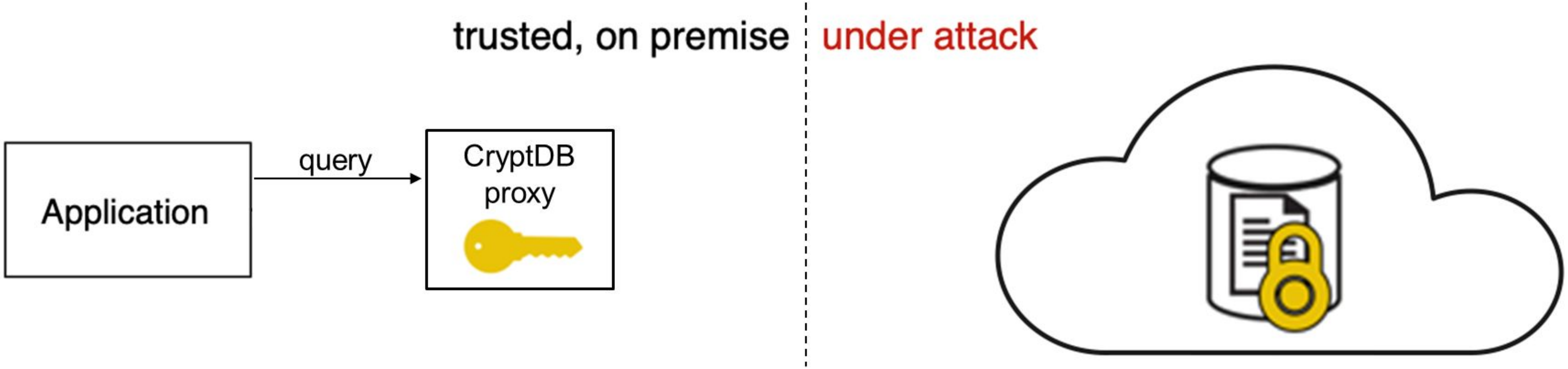
CryptDB (Popa11) was a first DBMS to process SQL queries on encrypted data.



[Popa11]

Encrypted Databases

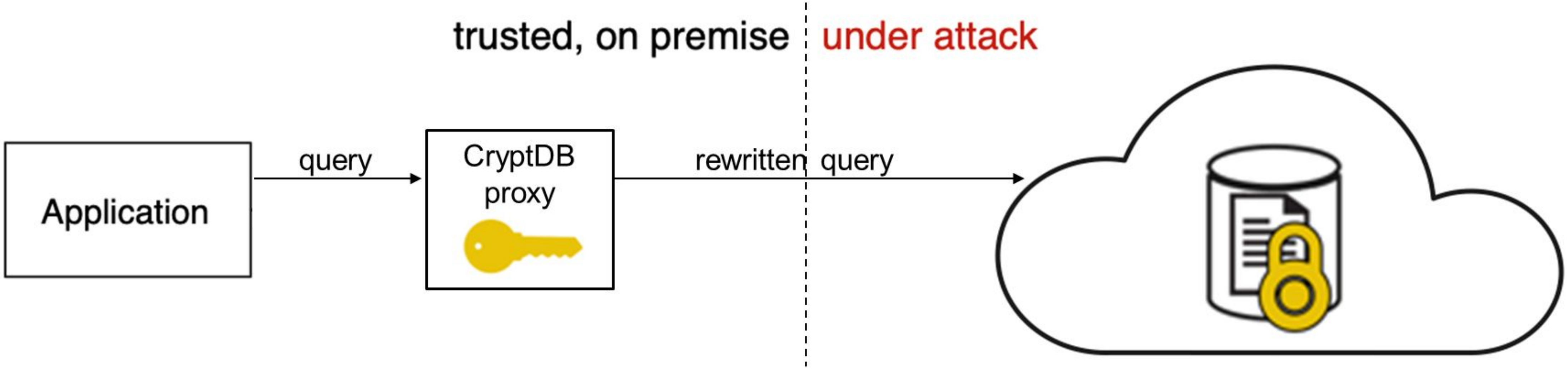
CryptDB (Popa11) was a first DBMS to process SQL queries on encrypted data.



[Popa11]

Encrypted Databases

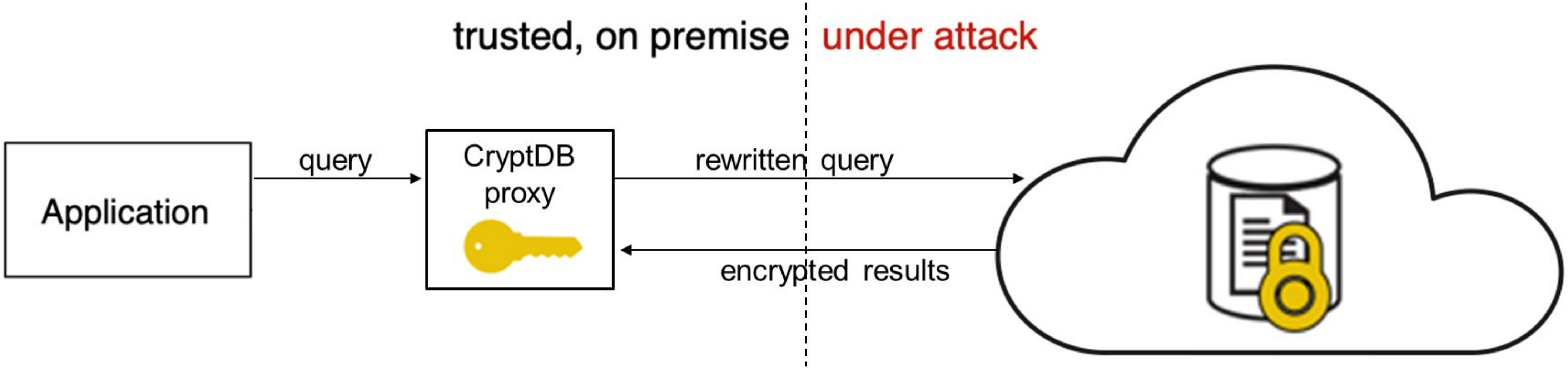
CryptDB (Popa11) was a first DBMS to process SQL queries on encrypted data.



[Popa11]

Encrypted Databases

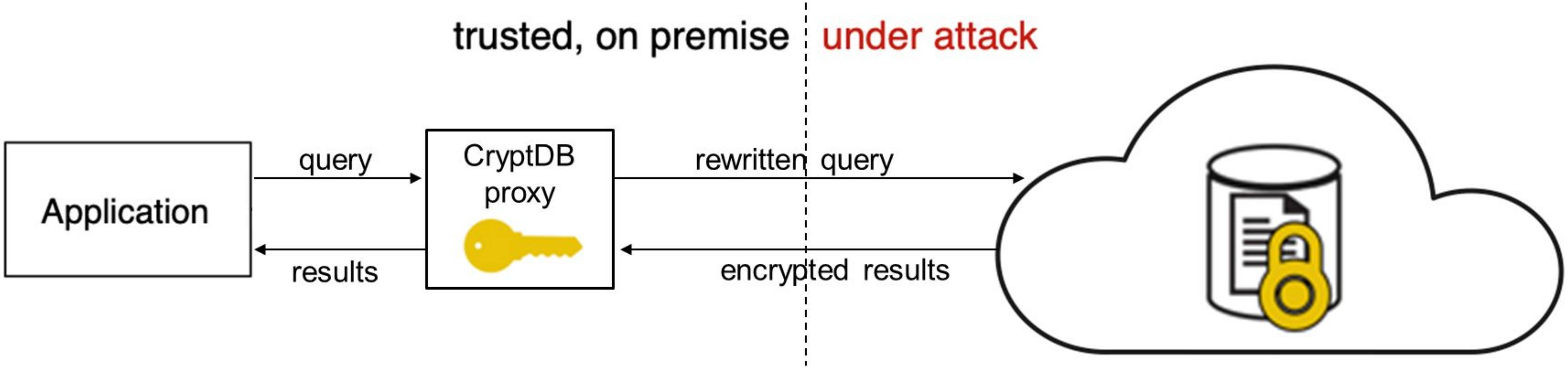
CryptDB (Popa11) was a first DBMS to process SQL queries on encrypted data.



[Popa11]

Encrypted Databases

CryptDB (Popa11) was a first DBMS to process SQL queries on encrypted data.



[Popa11]

CryptDB in a Nutshell

- Observation: most SQL can be implemented with a few operations (e.g., +, =, >)
- Methods:
  - Employs an efficient encryption scheme for each operation: Paillier for +; DET for =, order-preserving encryption for >, …
  - Maintains multiple ciphertexts of the data, one for each encryption
  - Redesigns the query planner to produce encrypted and transformed query plans, transparently for DBMS and applications
- Evaluation on TPC-C benchmarks shows 27% performance overhead

Existing Systems

- Academic
  - CryptDB
  - Cipherbase
  - Autocrypt
- Industry
  - Microsoft: AlwaysEncrypted
  - Google: EncryptedBigQuery
  - Skyhigh Security

Cited References

(Gentry09) Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, 2009.

(Popa11) Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. *In Proceedings of ACM Symposium on Operating Systems*, 2011.

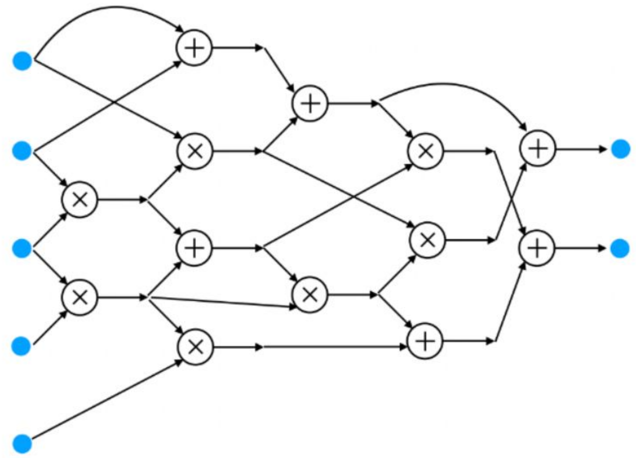
Example System: Homomorphic Databases

The End

Demo: HE/FHE in Practice

Concrete

- Rust implementation of TFHE [1]
- FHE based on Learning With Errors (LWE) hardness
- Boolean and arithmetic operations
    - Functions that can be compiled to circuits.
    - No arbitrary if/else statements or loops (why?)
- See notebook on Courseworks
    - Simple FHE circuits
    - Evaluating HE vs FHE runtime
    - Lightweight ML model inference on encrypted data



Source: Zama.ai

Learning with Errors (LWE): Basis

- LWE is a fundamental problem in lattice-based cryptography, a promising area for developing quantum-resistant cryptographic systems.
- It's based on the difficulty of solving a system of linear equations that have been intentionally perturbed by a small amount of "noise."
- **Problem:**
  - You have a secret vector **s** with **n** integers modulo *q* (where *q* is a large prime number). You are given a series of *m* equations (where $m>=n$). Each equation looks like this:

$$a_i \cdot s + e_i = b_i \pmod{q}$$

  where:

    - **a_i** is a publicly known vector of *n* integers modulo *q*. a_i are chosen uniformly at random.
    - **s** is the secret vector of *n* integers modulo *q* that we want to find (dot is the dot product).
    - **e_i** is a **small** integer "error" term, chosen from a specific probability distribution (often a discrete Gaussian distribution or a uniform distribution over a small range). The key is that **they are significantly smaller than *q*.**
    - **b_i** is the result of the noisy linear combination, an integer modulo *q*, also publicly known.
- You are given a collection of pairs **(a_i, b_i)** and the goal of the **search version** of the LWE problem is to recover the secret vector **s**.

Learning with Errors (LWE): Basis (cont.)

- LWE has been reduced to some **worst-case lattice problems** (shortest vector problem, closest vector problem), which are believed to be **hard even for quantum computers** (i.e., finding a solution requires exponential time in input size and quantum doesn't seem to give an edge).
- Intuition why it's hard:
  - **The Noise:** If the error term e_i were zero in all equations, then we would have a standard system of linear equations modulo q. Such systems can be efficiently solved using techniques like Gaussian elimination (modulo q). The introduction of the small, random errors makes this direct algebraic approach fail. The noise obscures the underlying linear relationship.
  - **The Modulo Operation:** Working modulo q (a finite field) adds another layer of complexity. Standard techniques for solving linear systems over real numbers or integers don't directly apply.
- For this reason, LWE is used as a basis for many FHE schemes. One example follows.

LWE-based FHE: Encryption

Encryption Formula

$$c = (a, b) = \left(a, a \cdot s + e + m \cdot \frac{q}{2}\right) \mod q$$

Parameter Descriptions

- q **(Modulus):** A large prime or power of two defining modular arithmetic.
- n **(Security Parameter):** Defines vector dimensions, impacting security. Secret key size, typical values: 512, 1024.
- s **(Secret Key):** A random vector $s \in \mathbb{Z}_q^n$, known only to the decryption party.
- a **(Public Random Vector):** Randomly sampled from $\mathbb{Z}_q^n$, included in ciphertext.
- e **(Error Term):** Small random noise (Gaussian-distributed) ensuring security.
- m **(Message Bit):** Plaintext (0 or 1) encoded as $m \cdot \frac{q}{2}$.
- b **(Second Ciphertext Component):** Encapsulates the secret key and error, making decryption non-trivial.

LWE-based FHE: Decryption

Decryption Formula

$$m' = \text{round}\left(\frac{(b - a \cdot s) \mod q}{q/2}\right)$$

Decryption Process

1. Compute $b - a \cdot s$ to remove the influence of the secret key:

$$b - a \cdot s = e + m \cdot \frac{q}{2} \mod q$$

2. Since e is small, the result is close to either **0** (if $m = 0$) or $q/2$ (if $m = 1$).
3. Apply rounding to extract m.

Key Considerations

- Decryption works only if **noise remains small**; too much noise results in incorrect rounding.
- Homomorphic computations increase noise, requiring **bootstrapping** for indefinite computation.

LWE-based FHE: Example Calculations

Addition of Two Ciphertexts

Given ciphertexts $c_1 = (a_1, b_1)$ and $c_2 = (a_2, b_2)$:

$$c_{\text{sum}} = (a_1 + a_2, b_1 + b_2) \mod q$$

- Result decrypts to $m_1 + m_2$, with noise growing as $e_1 + e_2$.

Multiplication of Two Ciphertexts

Multiplication increases noise significantly:

$$c_{\text{prod}} = (a_1 \cdot a_2, b_1 \cdot b_2) \mod q$$

- Noise grows **quadratically**, requiring bootstrapping after a few multiplications.

Addition example

We have two ciphertexts encrypting messages m_1 and m_2:

$$c_1 = (a_1, b_1) = \left(a_1, a_1 \cdot s + e_1 + m_1 \cdot \frac{q}{2}\right) \mod q$$

$$c_2 = (a_2, b_2) = \left(a_2, a_2 \cdot s + e_2 + m_2 \cdot \frac{q}{2}\right) \mod q$$

To add these ciphertexts, we compute:

$$c_{\text{sum}} = (a_{\text{sum}}, b_{\text{sum}}) = (a_1 + a_2, b_1 + b_2) \mod q$$

So, the resulting ciphertext after addition is:

$$c_{\text{sum}} = (a_1 + a_2, (a_1 + a_2) \cdot s + (e_1 + e_2) + (m_1 + m_2) \cdot \frac{q}{2}) \mod q$$

Addition example (cont)

The resulting ciphertext after addition was:

$$c_{\text{sum}} = (a_{\text{sum}}, b_{\text{sum}}) = \left(a_1 + a_2, (a_1 + a_2) \cdot s + (e_1 + e_2) + (m_1 + m_2) \cdot \frac{q}{2}\right) \mod q$$

The decryption process involves the following formula:

$$m' = \text{round}\left(\frac{(b_{\text{sum}} - a_{\text{sum}} \cdot s) \mod q}{q/2}\right)$$

Plug in the values:

$$b_{\text{sum}} - a_{\text{sum}} \cdot s = \left((a_1 + a_2) \cdot s + (e_1 + e_2) + (m_1 + m_2) \cdot \frac{q}{2}\right) - (a_1 \cdot s + a_2 \cdot s)$$

$$b_{\text{sum}} - a_{\text{sum}} \cdot s = (e_1 + e_2) + (m_1 + m_2) \cdot \frac{q}{2}$$

Now we can plug this back into the decryption formula:

$$m' = \text{round}\left(\frac{(e_1 + e_2) + (m_1 + m_2) \cdot \frac{q}{2}}{q/2}\right)$$

Addition example (cont)

$$m' = \text{round}\left(\frac{(e_1 + e_2) + (m_1 + m_2) \cdot \frac{q}{2}}{q/2}\right)$$

- Since e_1 and e_2 are small noise terms, their impact on the rounding is negligible compared to $(m_1 + m_2) \cdot \frac{q}{2}$.
- When rounding, the term $(m_1 + m_2) \cdot \frac{q}{2}$ dominates, leading to:

$$m' \approx m_1 + m_2$$

THUS:

- The decryption works correctly as long as the noise e1+e2 remains small relative to q/2.
- This highlights the robustness of LWE-based FHE for addition, as the structure of the encryption ensures the correct recovery of the original message.

Demo: Concrete

Notebook on courseworks.

Cited References

[1] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast Fully Homomorphic Encryption Over the Torus," J Cryptol, vol. 33, no. 1, pp. 34–91, Jan. 2020, doi: 10.1007/s00145-019-09319-x.

The End

Homework 3 Overview

(CA walks through HW3 notebook, posted on Courseworks)