11/4/26

COMP 435: *Computer Security Concepts*

Lecture 28: Digital Signatures, Intro to Software Security



The University of North Carolina at Chapel Hill

https://tinyurl.com/comp435-fa26

Logistics



- Lab 3 due tomorrow
  - Due 11/5

Plan for Class Today

- 10:10-10:25 – Digital Signatures
- 10:35- 10:50 – Common Software Security Vulnerabilities and Best Practices

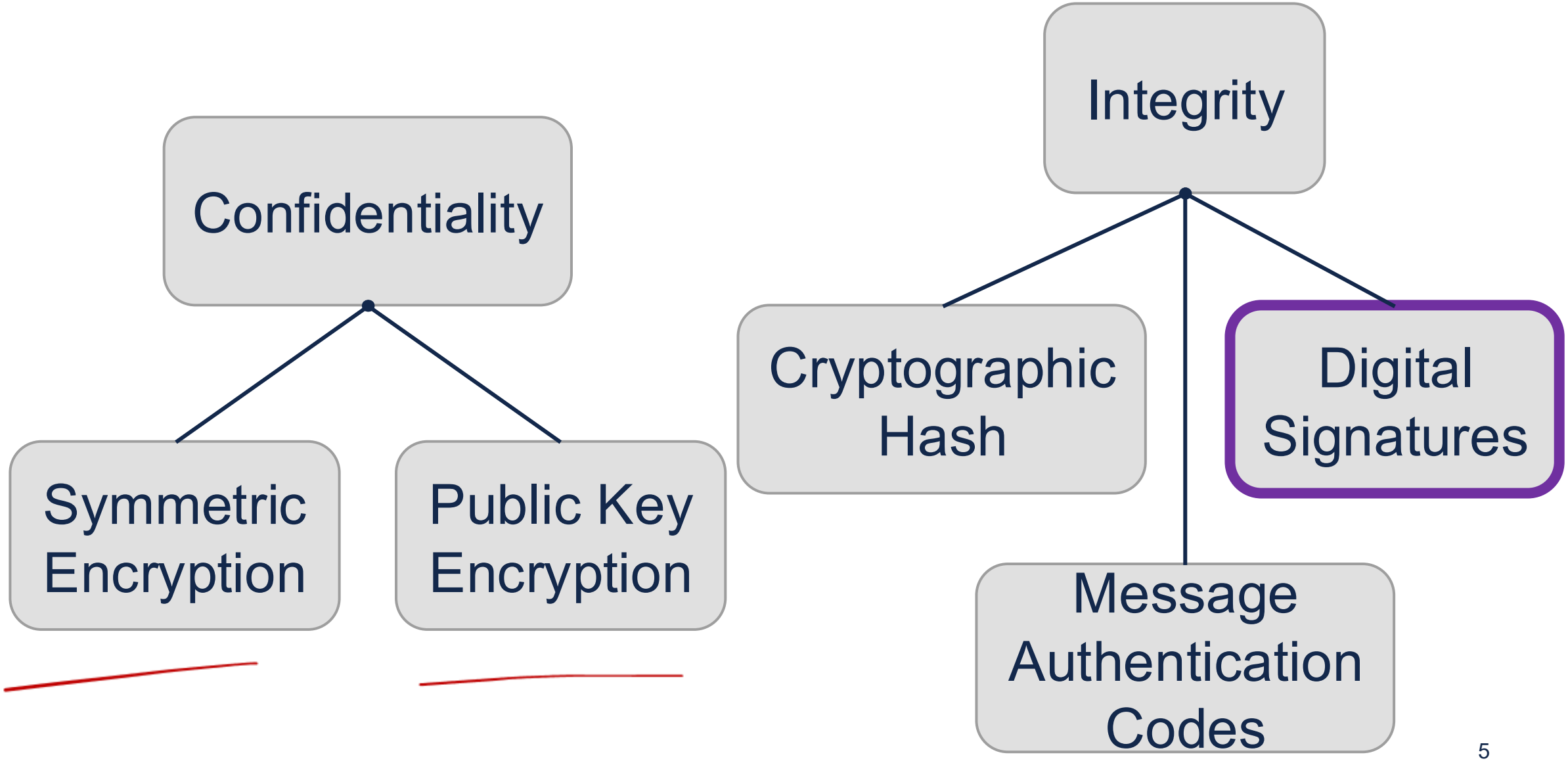
Guiding Question ☺

What cryptographic tool(s) give us non-repudiation (aka accountability)?



Digital Signatures

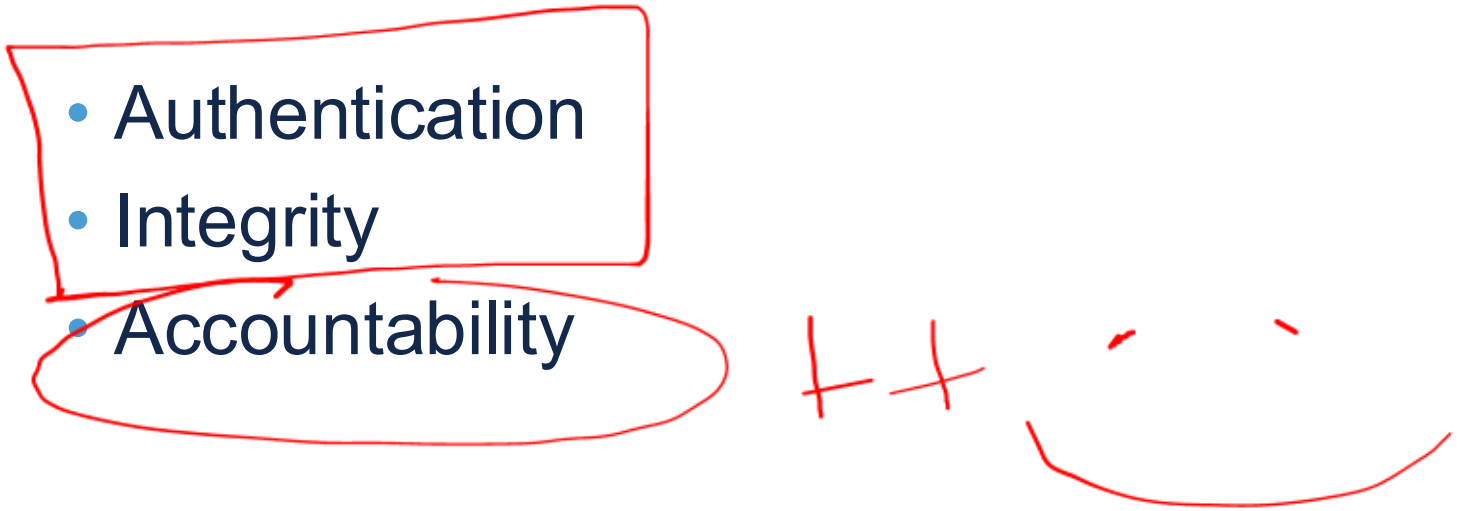
Roadmap


Confidentiality
Symmetric Encryption
Public Key Encryption
Integrity
Cryptographic Hash
Digital Signatures
Message Authentication Codes

Digital Signature

Def’n: a means to ensure a given message comes from a given source.

Properties of a Digital Signature



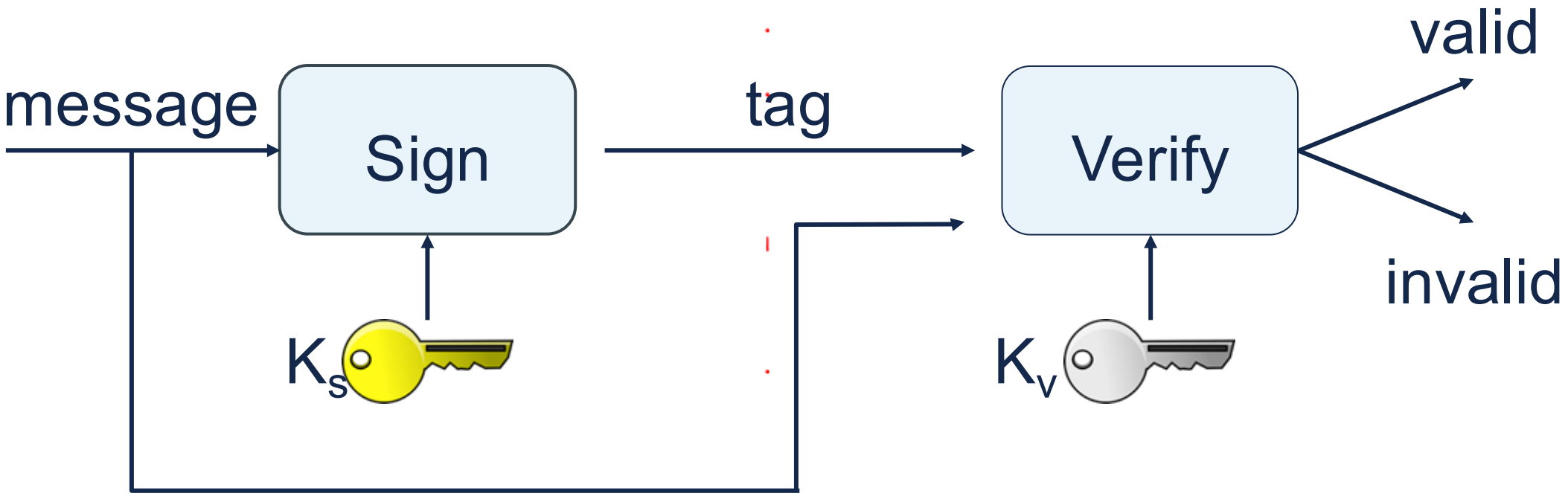
- Authentication
- Integrity
- Accountability

Digital Signature Requirements

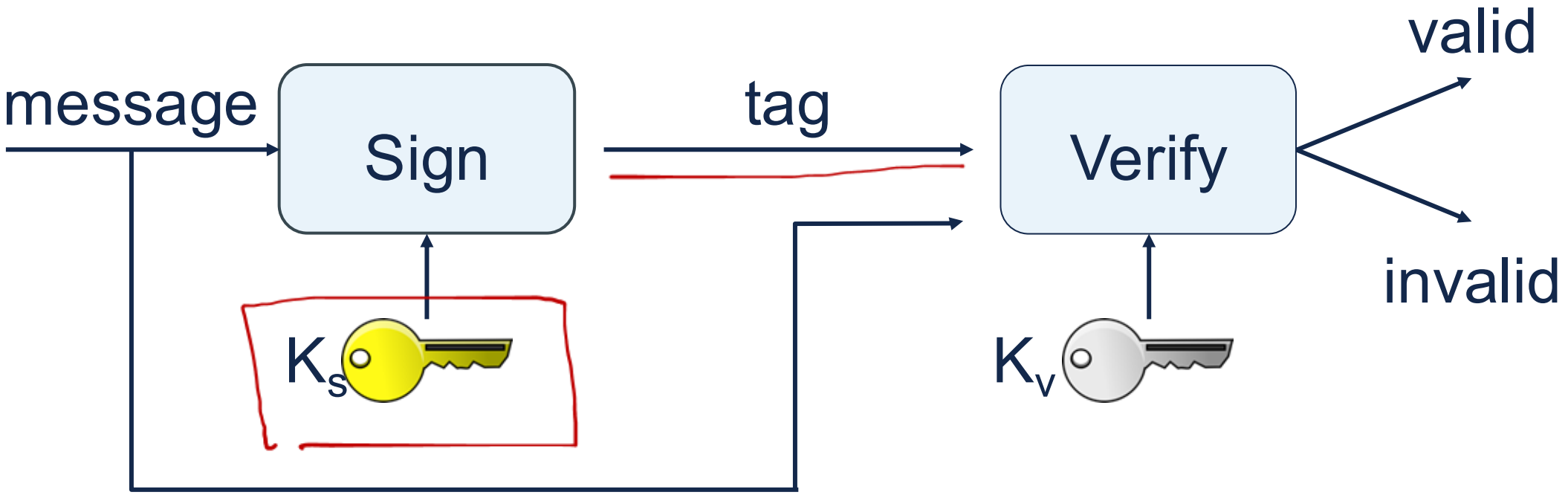
- Unforgeable
- Authentic
- Unalterable
- Not reusable

Public Key Signatures

public key crypto



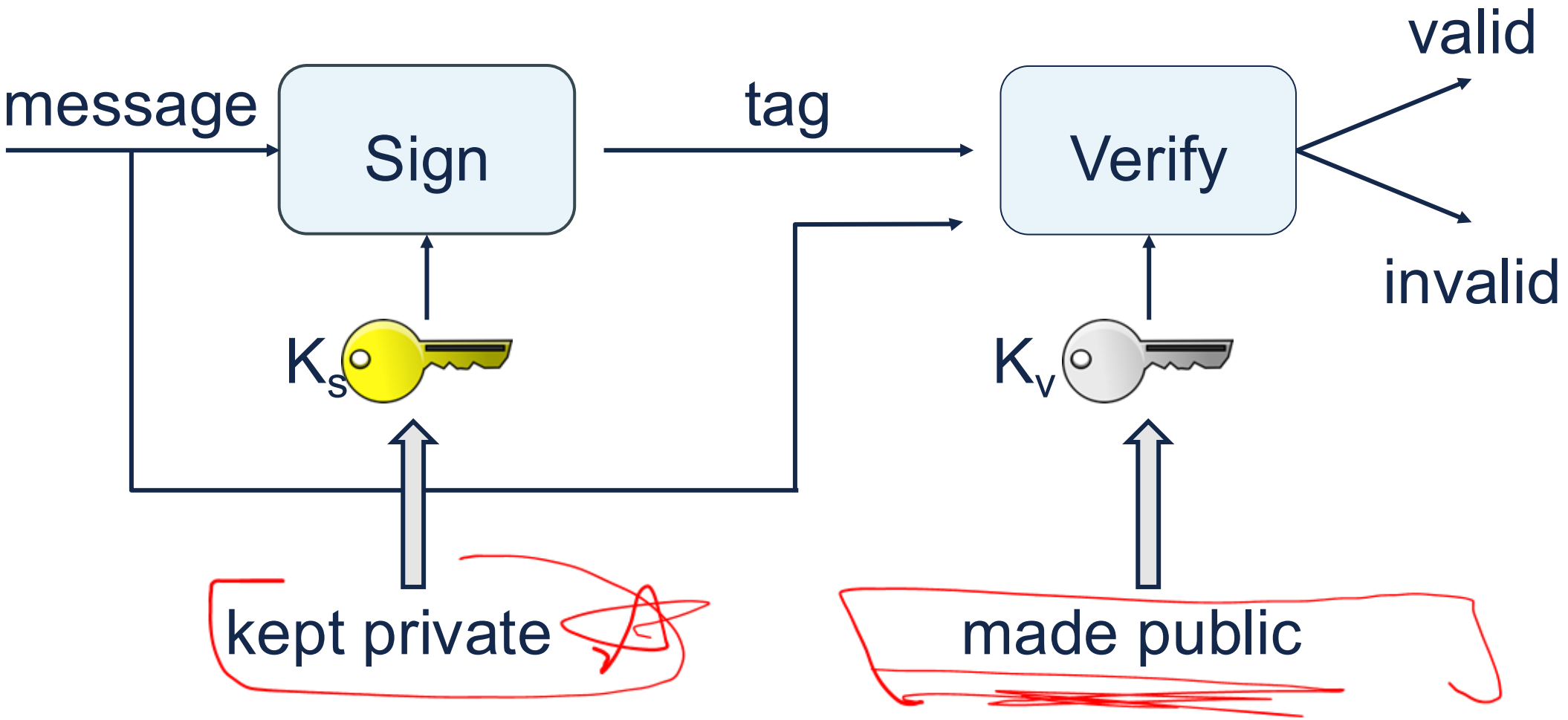
Public Key Signatures

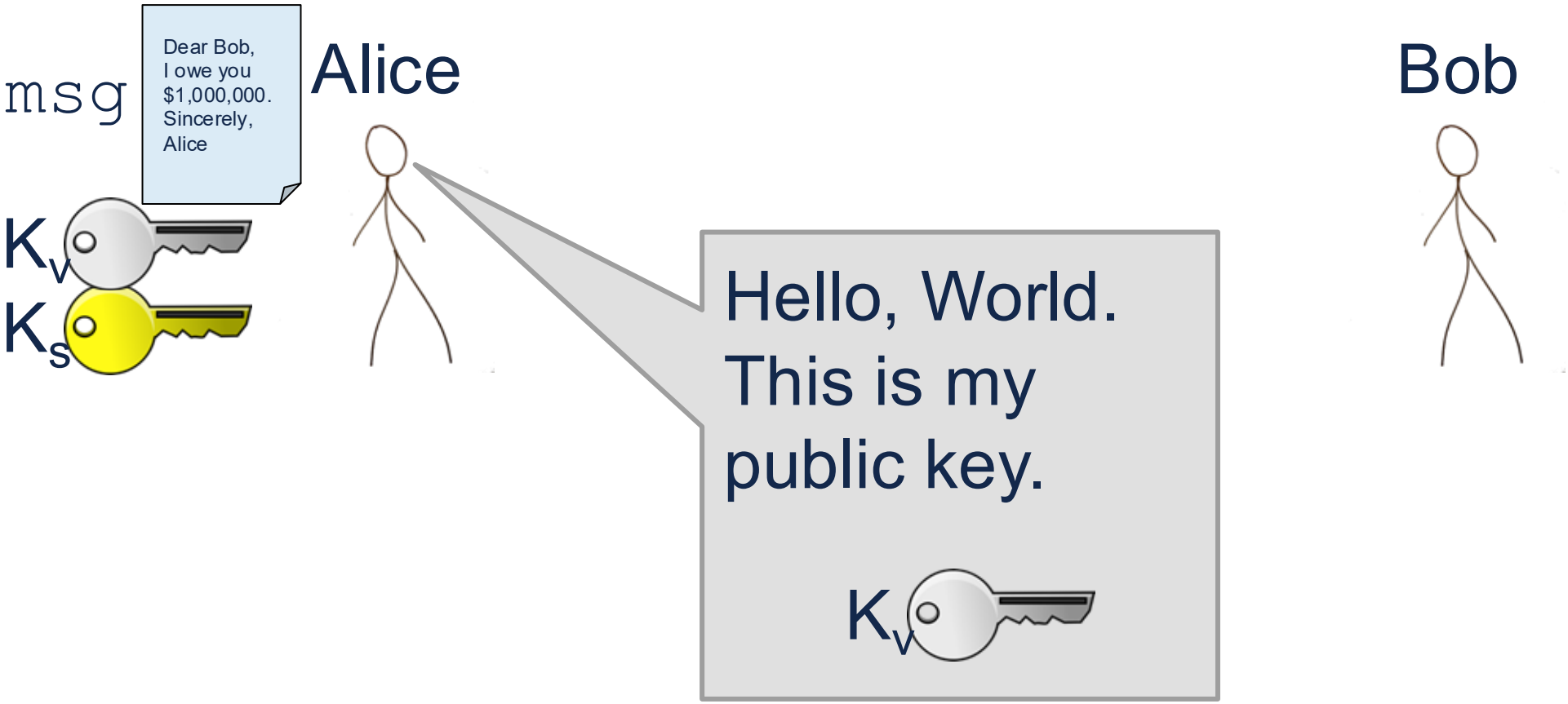


$$tag = Sign_{Ks}(msg)$$

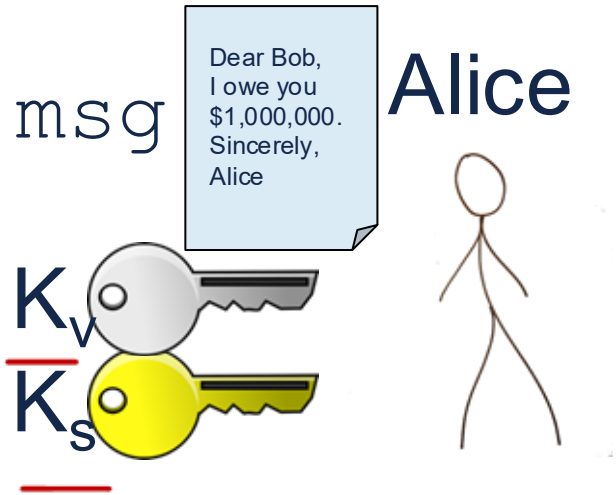
$$isValid = Verify_{Kv}(msg, tag)$$

Public Key Signatures




msg
Dear Bob,
I owe you
$1,000,000.
Sincerely,
Alice
Alice
Bob
K_v
K_s
Hello, World.
This is my
public key.
K_v

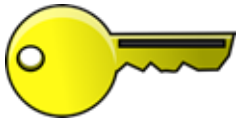
msg



Bob



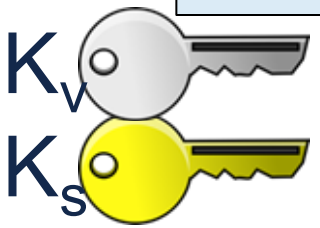
$tag := Sign_{Ks}(msg)$



msg

Dear Bob,
I owe you
$1,000,000.
Sincerely,
Alice

Alice



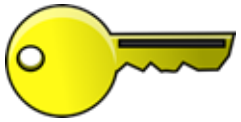


Bob





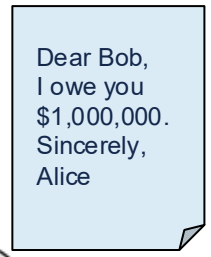
$tag := Sign_{Ks}(msg)$



(tag, msg)

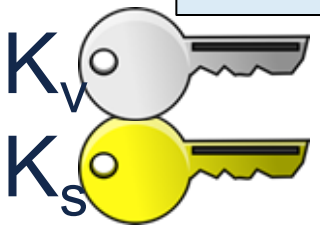


msg



Alice

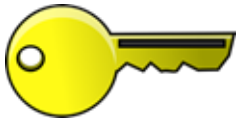




Bob



$tag := Sign_{Ks}(msg)$



(tag, msg)



$isValid = Verify_{Kv}(msg, tag)$





RSA for Digital Signatures

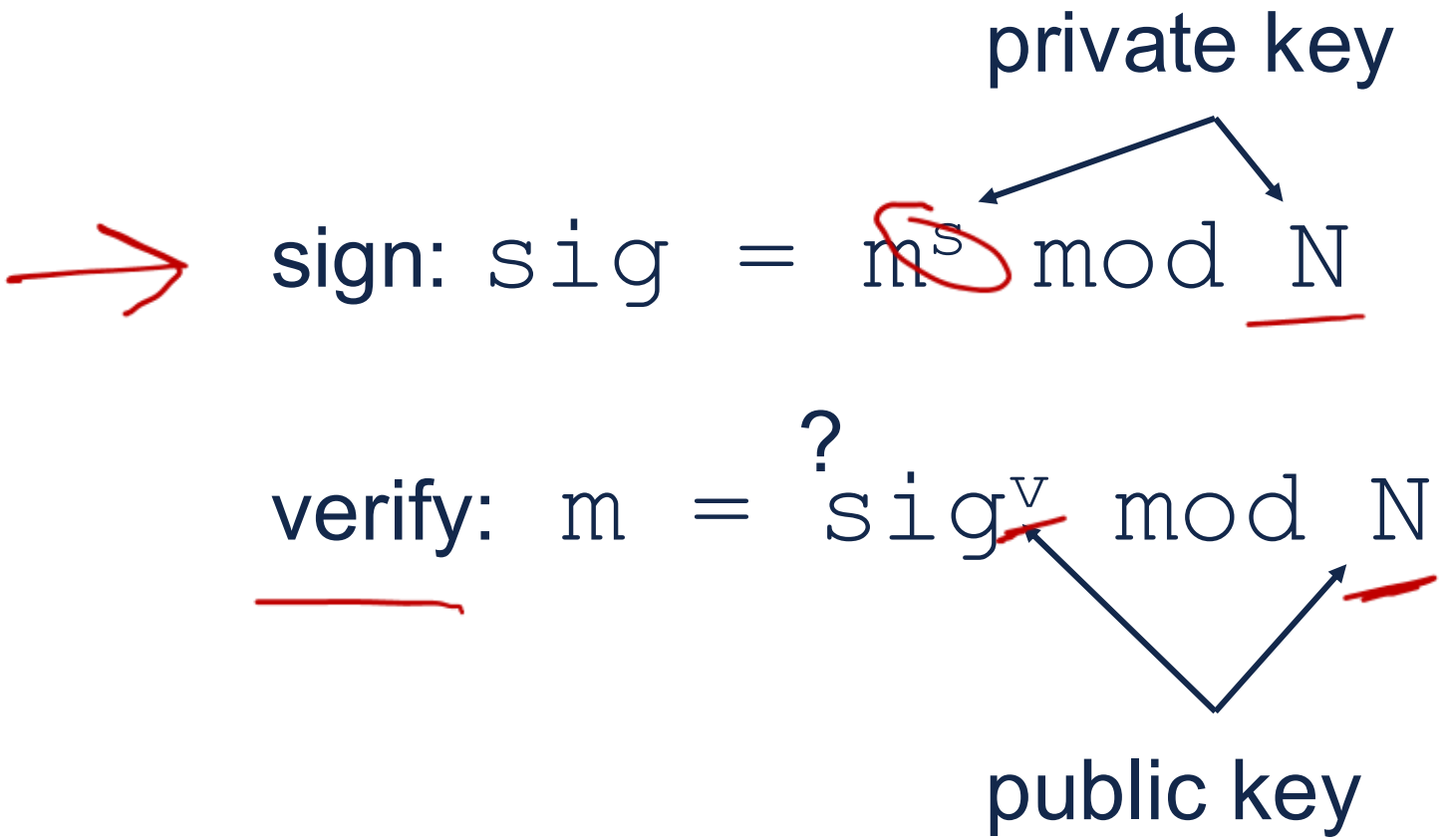
RSA for digital signatures



sign: $sig = m^s \bmod N$

verify: $m \stackrel{?}{=} sig^v \bmod N$

RSA for digital signatures

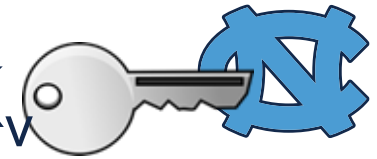


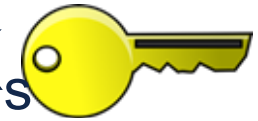
m_1, m_2

$m_1 \neq m_2$

$m_1 - m_2 \equiv (m) \bmod N$



(N, v) → K_v 

(N, s) → K_s 

$s_1 = m_1^s$

$s_2 = m_2^s$

$s_1 \cdot s_2 = (m_1^s)(m_2^s)$



target :=

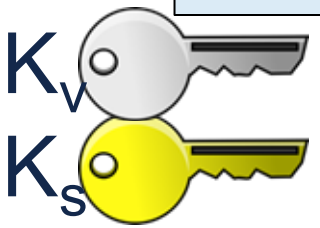
```
sig := Rand(N)
msg := sig^v mod N
```

sig1, sig2 for m_1, m_2

RSA does not provide unforgeability

msg

Dear Bob,
I owe you
$1,000,000.
Sincerely,
Alice



Alice



Hello, World.
This is my
public key.



Bob

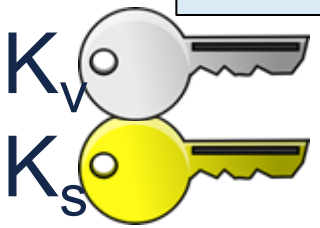


msg

Dear Bob,
I owe you
$1,000,000.
Sincerely,
Alice

Alice

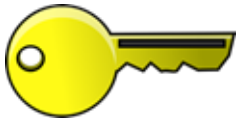




Bob



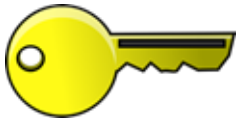
$tag := Sign_{K_S}(h(msg))$







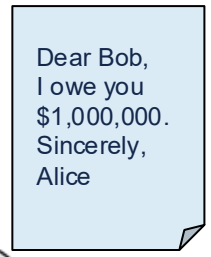
$tag := Sign_{K_S}(h(msg))$



(tag, msg)

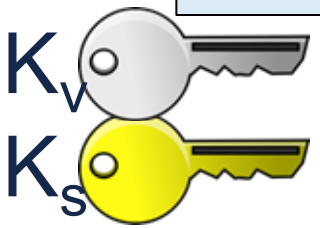


msg



Alice



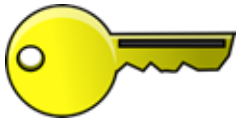


Bob





$tag := Sign_{Ks}(h(msg))$





(tag, msg)



$isValid = Verify_{Kv}(h(msg), tag)$



WS Q2

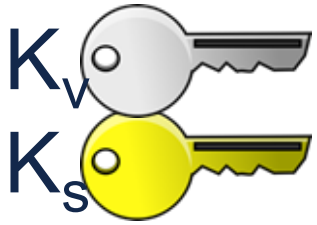
2.1

2.3 No confidentiality

msg sent in pt.



Encrypt then Sign



Alice

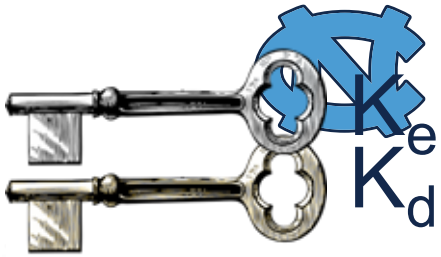


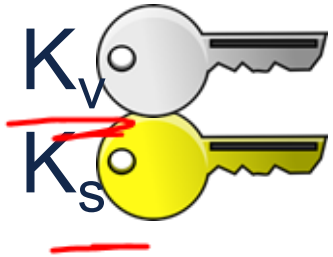
Hello, World. This is my public key for verifying my signature.



Bob



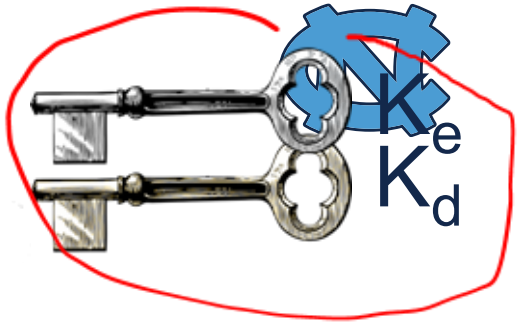




Alice



Bob

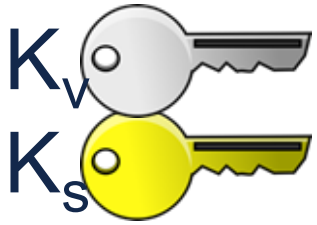




Hello, World. This is my public key for encrypting.



K_e



Alice

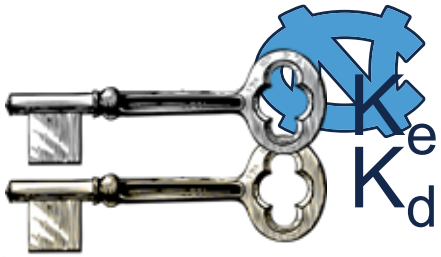


msg

Dear Bob,
I owe you
$1,000,000.
Sincerely,
Alice

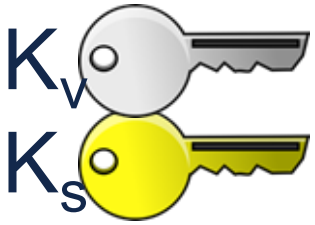
Bob





$c := Enc_{Ke}(msg)$

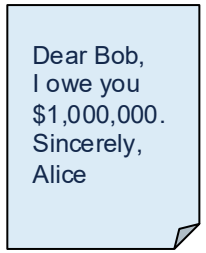
$tag := Sign_{ks}(c)$



Alice

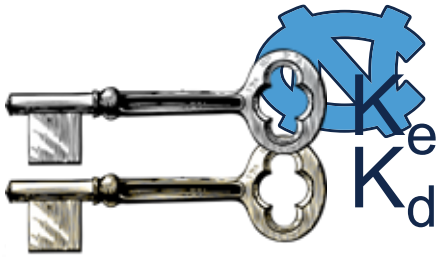


msg



Bob



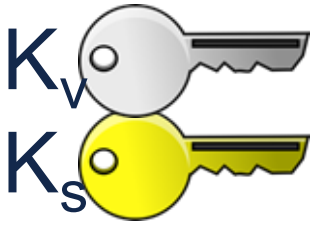


$c := Enc_{Ke}(msg)$

$tag := Sign_{ks}(c)$

(c,tag)

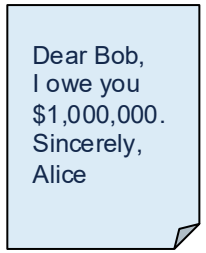




Alice

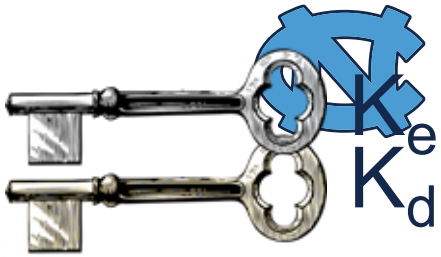


msg



Bob





$c := Enc_{Ke}(msg)$

$tag := Sign_{ks}(c)$

(c,tag)



$isValid := Verify_{Kv}(c,tag)$

$msg := Dec_{kd}(c)$

Old WS Q3 (lec 26)



3.1 - Bob's public key

- Alice's private

- Bob's verified w/ Alice pub

Bob decrypt w Bob's private

3.2 Sruce fail

A Note of Caution



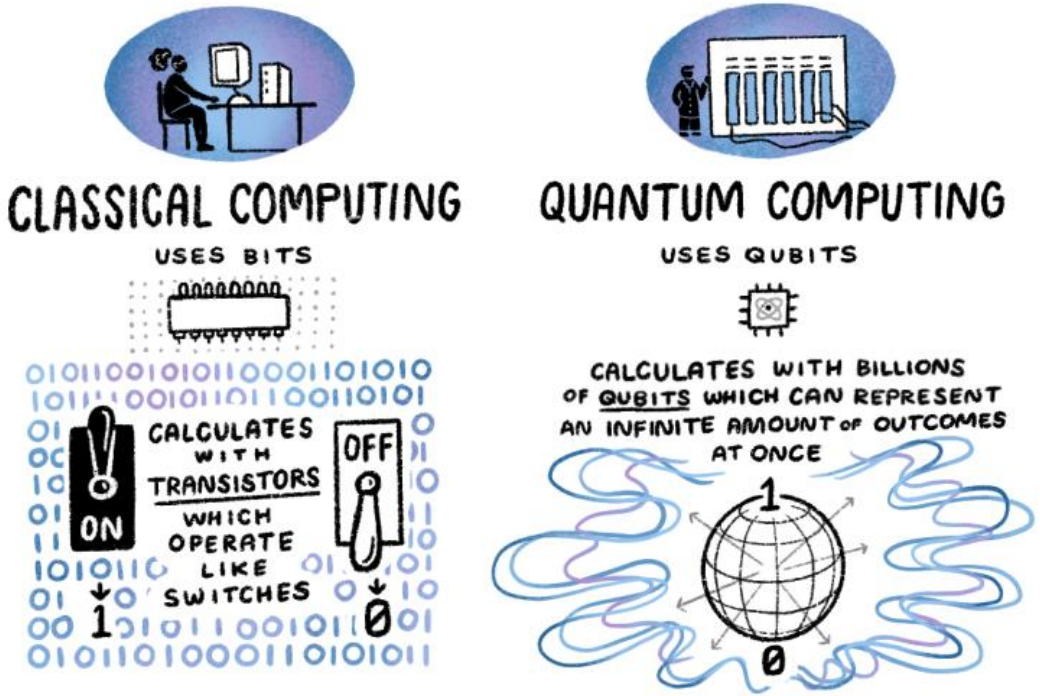
- Beware of determinism
- Use unique keys for encryption, signing, generating MACs
- Use appropriate algorithms for the desired property
- Do not roll your own

Before we move on… what about quantum?

- Maybe you noticed that quiz question :)

What is a quantum computer?

- Uses **qubits**, which can be in a superposition of 0 and 1. measuring a qubit gives 0 or 1.
- Quantum algorithms use interference to make useful outcomes more likely.



https://blockclubchicago.org/2025/02/06/what-is-quantum-computing/

Why do we care?

- Can solve **certain problems much faster** than classical computers.
- **Why do we care??** Shor's algorithm could break RSA and Diffie-Hellman on a sufficiently powerful quantum computer.
  - That affects key establishment and digital signatures.
  - Ex: Recorded encrypted traffic could be decrypted later. ☹

Recall: RSA Security



- $c = msg^e \bmod N$
- The attacker's challenge:
  Given N, e, msg^e, find msg
- If the attacker can factor N, they win

Recall: DH Security



- $K = A^b = (g^a)^b = (g^b)^a = B^a \bmod p$
- The attacker's challenge:
  - given g, p, A, B,
  - find K
- If the attacker can calculate the discrete log of A or B, they win

Recall: One-Way Functions



- a function F: X → Y is one-way if F is easy to compute but hard to invert

- multiplying two large primes is one-way
  - multiplying pq = N is easy to compute
  - factoring N to find p, q is hard

Recall: One-Way Function



- a function F: X → Y is one-way if F is easy to compute but hard to invert
- modular exponentiation is one-way
  - computing $g^a \bmod p = A$ is easy
  - computing $\text{dlog}_{g,p}(A) = a$,
    the discrete log of A mod p with base g, is hard

Enter: Schorr's Algorithm!!

- "Hard to reverse" depends on the computing model.
- Shor's algorithm efficiently factors integers and solves discrete logarithms on a quantum computer.
- This undermines the hardness assumptions behind RSA and Diffie–Hellman.
- Requires a sufficiently large, reliable quantum computer.

Post-quantum replacements



- ML-KEM: establish a shared secret (NIST FIPS 203).
- ML-DSA: create digital signatures (NIST FIPS 204).
- SLH-DSA: another signature option (NIST FIPS 205).

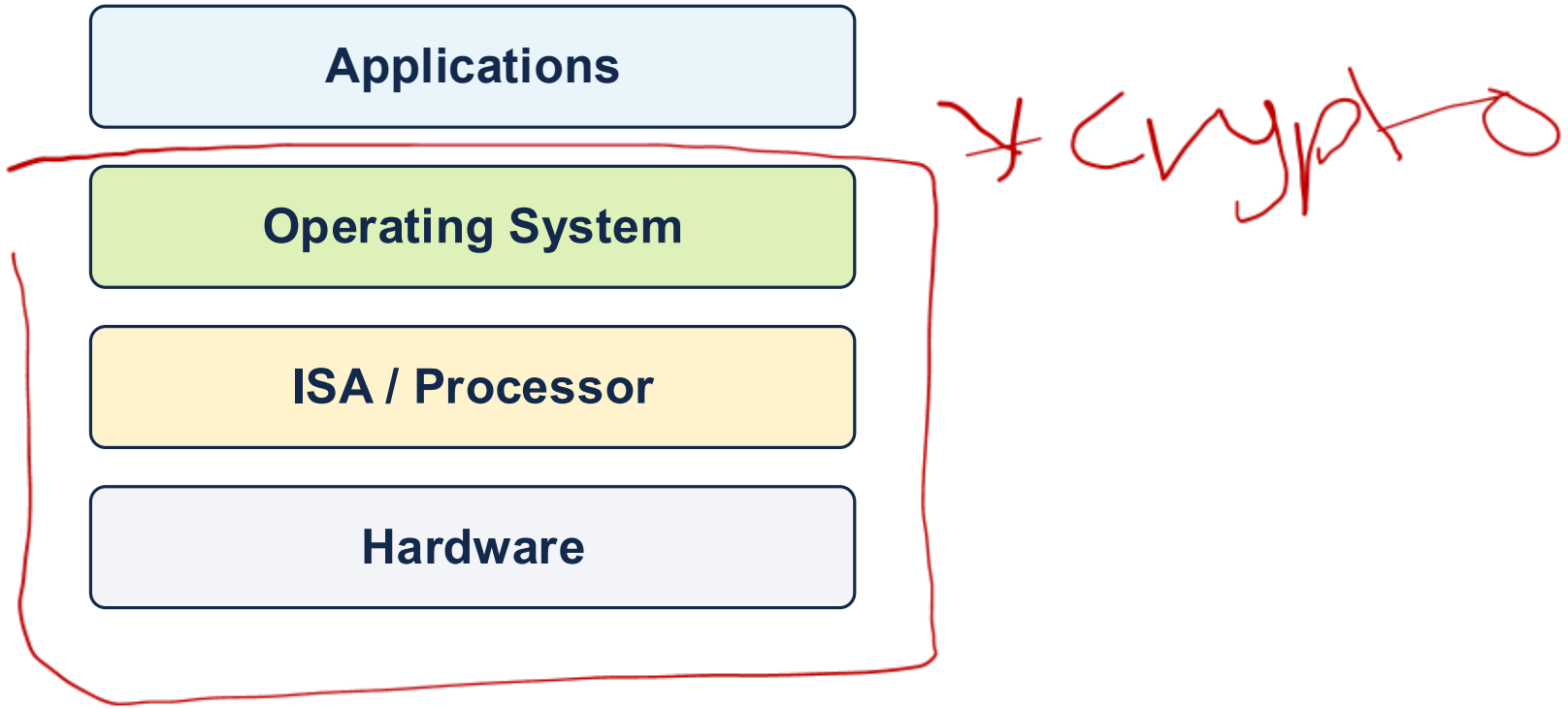
Take cryptography for more! ☺



Software Security: Vulnerabilities

The Stack We Are Trusting

- Every software security mechanism rests on assumptions about the layers below it.





1. Integer Overflow

> INT_MAX_VALUE



unsigned int

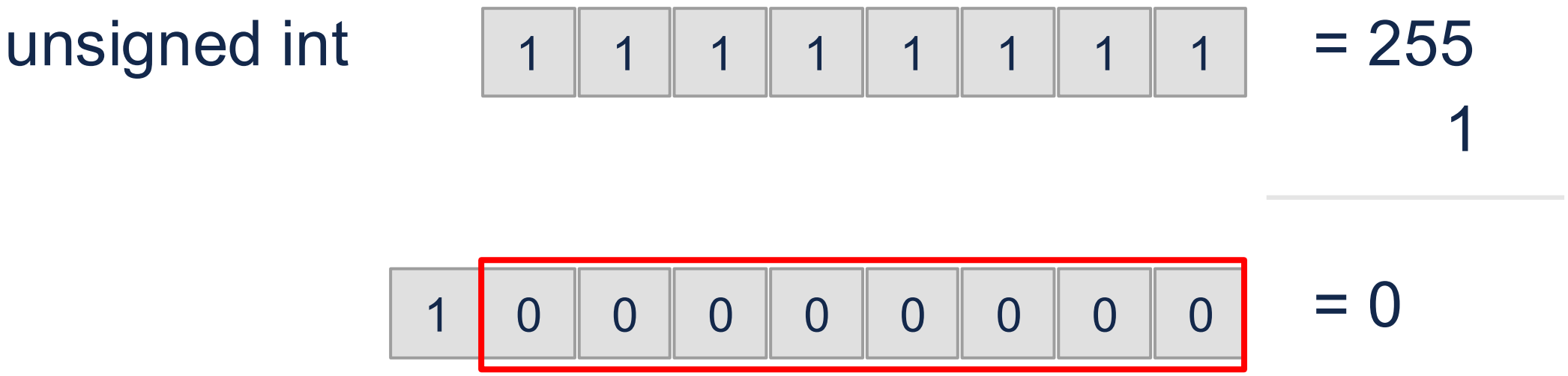
1	1	1	1	1	1	1	1

= 255

1

1	0	0	0	0	0	0	0	0

= 256


unsigned int
1 1 1 1 1 1 1 1
= 255
1
1 0 0 0 0 0 0 0 0
= 0

```
1. int handle_login(userid, passwd) {
2.     attempts = attempts + 1; //4-bit signed int
3.     if (attempts <= 6) { //max attempts allowed is 6
4.         if passwdValid(userid, passwd) {
5.             attempts = 0;
6.             return (1);  //access granted
           }
       }
7.     return (0); //what happens on the next attempt? }
```

With a partner, trace attempts 6, 7, and 8. Does the program check the password each time? Then fix the code.

```
1. int handle_login(userid, passwd) {
2.   if (attempts >= 6) {
3.      return (0);
4.    }

5.    attempts = attempts + 1;

6.     if passwdValid(userid, passwd) {
7.        attempts = 0;
8.        return (1);
       }
    }
9.  return (0);
}
```





```
1. int *vulnerable(int *inparray, int len) {
2.   int *myarray;
3.   myarray = malloc(len * sizeof(int));
4.   if (myarray == NULL) {
5.     return (-1);
     }
6.   for (i = 0; i < len; i++) {
7.     myarray[i] = inparray[i];
     }
8.   return myarray;
9. }
```

4 bytes

Line 3: memory for the array of integers is created.
Call to malloc says: allocate of memory equal to len * size of an int

Ex: an int is 4 bytes and if the length is 5, this will allocate 20 xbytes

```
1. int *vulnerable(int *inparray, int len) {
2.   int *myarray;
3.   myarray = malloc(len * sizeof(int));
4.   if (myarray == NULL) {
5.     return (-1);
     }
6.   for (i = 0; i < len; i++) {
7.     myarray[i] = inparray[i];
     }
8.   return myarray;
9. }
```



Line 3: memory for the array of integers is created.
Call to malloc says: allocate of memory equal to len * size of an int

Problem: if the length is very big (ex., close to MAX_INT), if we multiply this by 4, we will **overflow** and allocate a **small number**

```
1. int *vulnerable(int *inparray, int len) {
2.    int *myarray;
3.    myarray = malloc(len * sizeof(int));
4.    if (myarray == NULL) {
5.      return (-1);
      }
6.    for (i = 0; i < len; i++) {
7.      myarray[i] = inparray[i];
      }
8.    return myarray;
9. }
```

Lines 6/7 copy up to "len" values, but space actually allocated for `myarray` is potentially very small if we overflow

Problem: memory past the end of the malloc'ed `myarray` will be overwritten

```
1. int *vulnerable(int *inparray, int len) {
2.    int *myarray;
3.    myarray = malloc(len * sizeof(int));
4.    if (myarray
5.       return (-1
      }
6.    for (i = 0;
7.       myarray[i]
      }
8.    return myarray;
9. }
```

Integer overflow being used as stepping stone for buffer overflow attack!

Lines 6/7 copy up to “len” values, but space actually allocated for `myarray` is potentially very small if we overflow

Problem: memory past the end of the malloc’ed `myarray` will be overwritten

Some reflections



- Catching these errors through manual review is very hard!
  - On first read through, this example seemed like it was doing the right checks!
- Integer underflow can also cause problems
  - Underflow results in a negative value → converted to an unsigned malloc parameter would be interpreted as very large unsigned int.
  - Attempt to allocate huge memory blocks could trigger out-of-memory conditions or return NULL pointers