# Advanced Internet Security
## VU 183.222

Adrian Dabrowski, Markus Kammerstetter, Johanna Ullrich, Georg Merzdovnik, Stefan Riegler,

Aljosha Judmayer

inetsec@seclab.tuwien.ac.at

# Introduction to
# *Cryptographic Engineering (II)*

# How to read these slides?



The material presented on the slides does sometimes contain more information than is necessary to know from the top of your head for the exam, but might be relevant for some challenges or useful background knowledge or recap

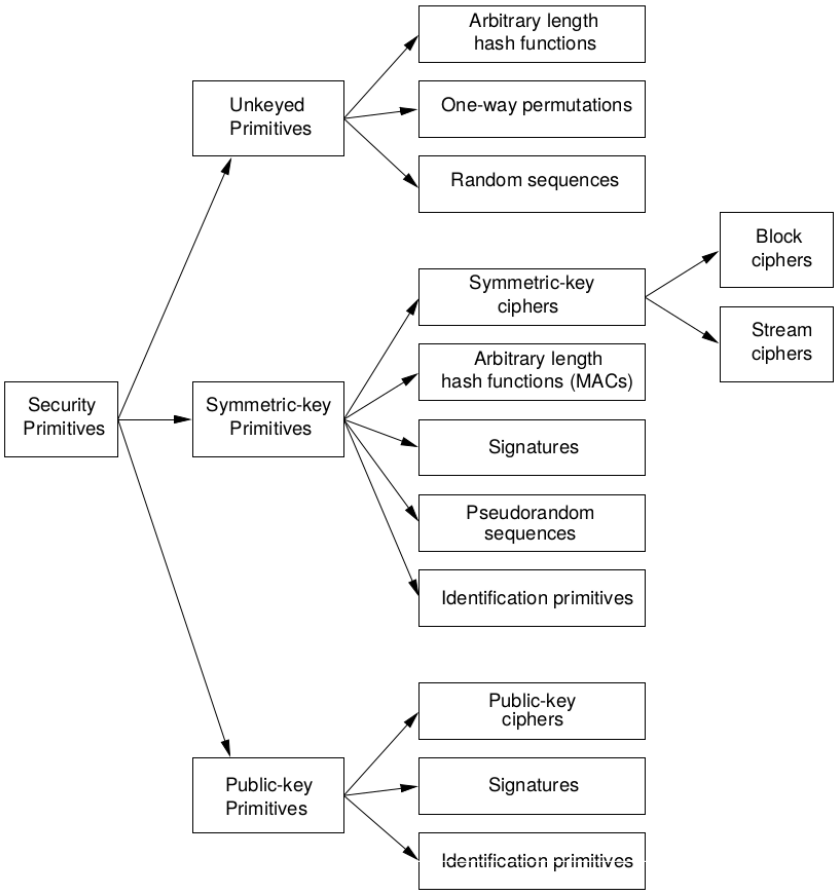
**Slide Log Levels:**

- **[WARN]**
  - This is highly important to understand and know in detail, also for the exam
- None
  - The default, potentially relevant for exam in shape of comprehension questions. Not required to be memorized 1:1 for the exam
- **[INFO]**
  - Recap or background information not required to know from the top of our head for the exam.
  - Might though be relevant for challenges

# Cryptographic Primitives



- **Unkeyed primitives**
  - hash functions
  - (real) random sequences
- **Symmetric-key primitives**
  - symmetric key ciphers
    - block ciphers
    - stream ciphers
  - Message Authentication Codes
  - signatures
  - pseudo-random sequences
- **Public-key primitives**
  - public-key ciphers
  - signatures




[1] http://cacr.uwaterloo.ca/hac/

# Cryptographic/Security Engineering



- 

- Agenda for today and key takeaways for today:
  - Misue of unkeyed primitives
    - Don’t use unkeyed primitives as symmetric-key primitives
      - i.e., Don’t use plain hash functions as MACs
    - Brute force attack strategy against large number of hashes
  - Textbook crypto != production crypto
  - **Nonce** stands for **n**umber that can **onl**y be used on**ce**

# Hash functions

# Hash length extension attack

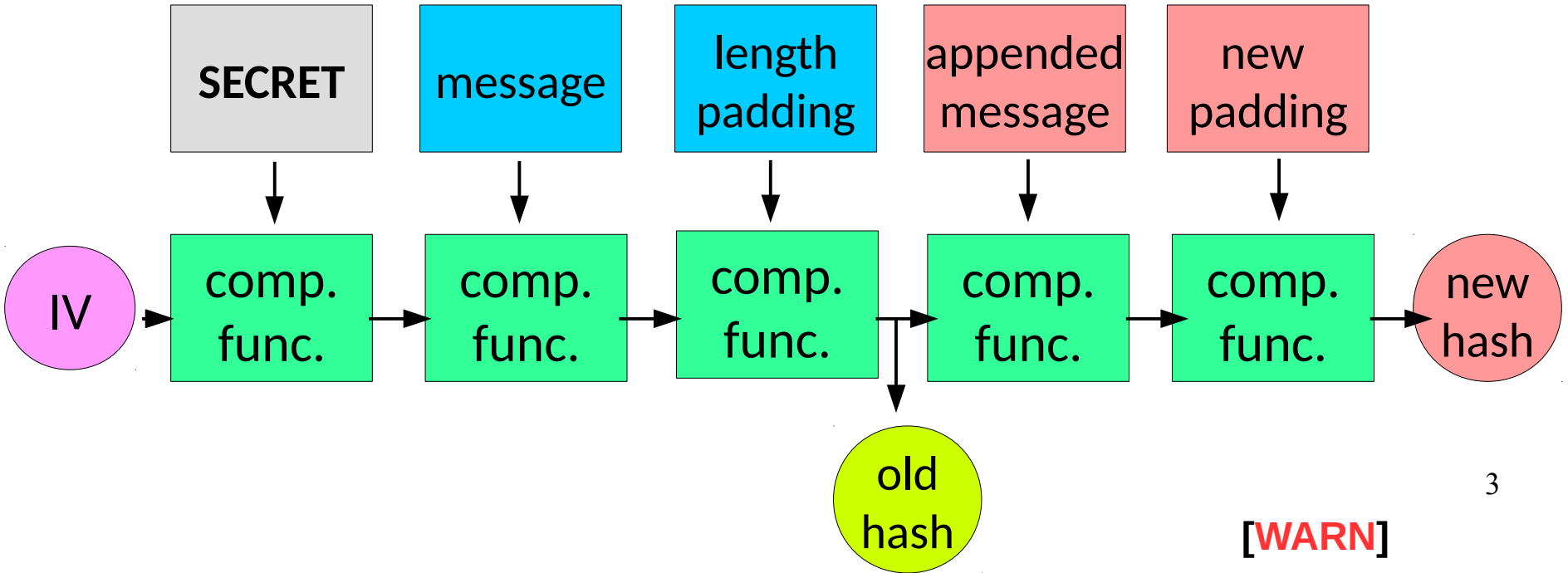
- In a hash length extension attack an attacker can use `H(message_1)` and the length of `message_1` to calculate `H(message_1||message_2)` for a `message_2` chosen by the attacker.
- All algorithms based on the Merkle-Damgard construction are susceptible to this kind of attack
  - e.g., MD5, SHA-1, SHA-2
- That does not mean that these algorithms are broken! Just that you should not use them directly as MAC (Message Authentication Code) to ensure the integrity and authenticity of some message



[WARN]

# Hash length extension attack

- *Merkle-Damgaard* construction
  - chain of compression functions and Initialization Vector
  - used in MD5, SHA1, SHA256, SHA512

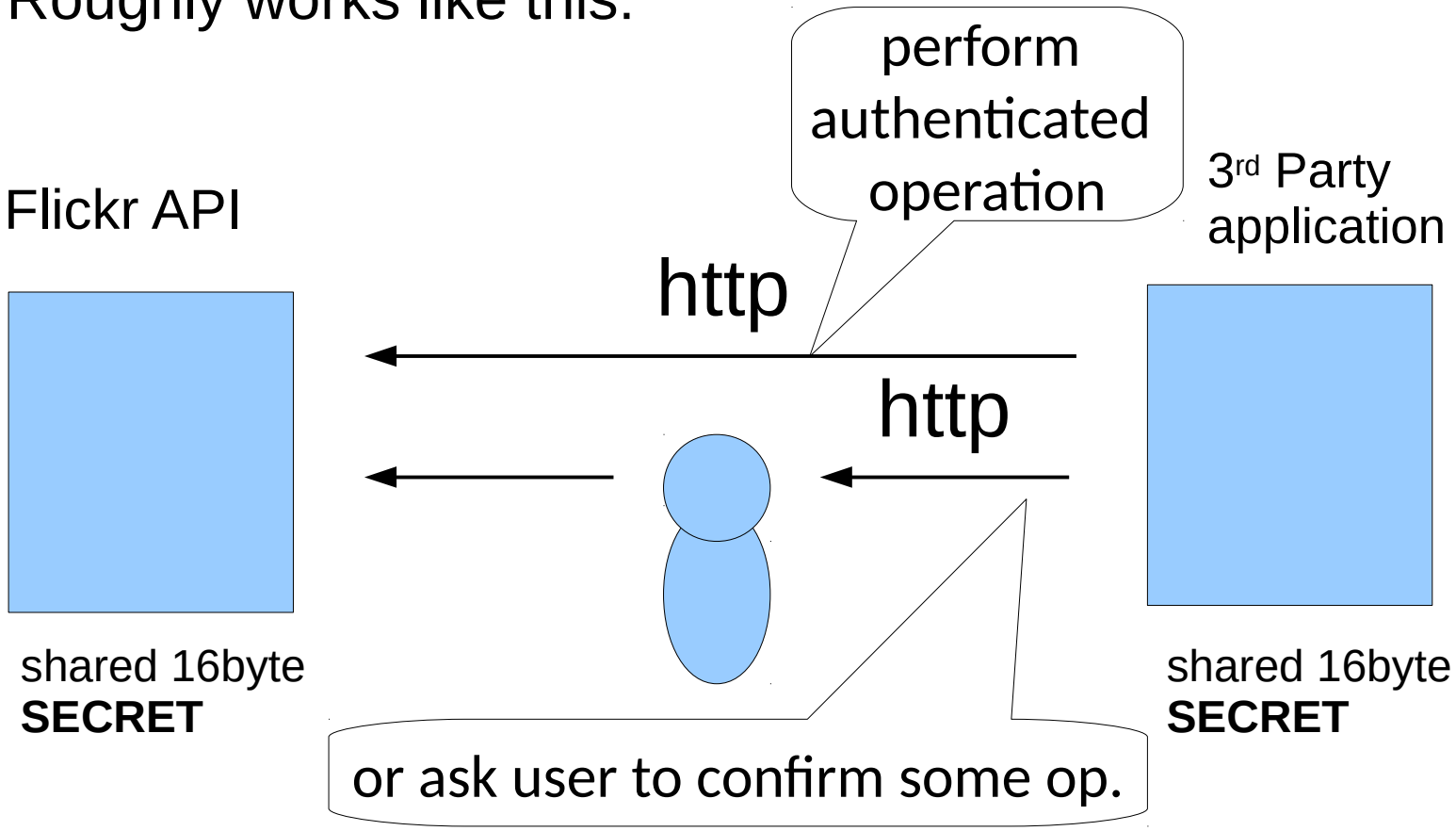





[WARN]

# Hash length extension attack

- Flickr's API Signature Forgery Vulnerability [1]
- Roughly works like this:






[1] http://netifera.com/research/flickr_api_signature_forgery.pdf

# Hash length extension attack

- Request:

```
http://www.flickr.com/services/auth/?
api_key=44fefa051fc1c61f
&perms=read
&api_sig=1a947ced7375aadf970ccd7ee2bb792b81d5ed1c
```

- Resulting hash function input:

```
 SECRETapi_key44fefa051fc1c61fpermsread
```

- Resulting output:

```
$ echo -n "SECRETapi_key44fefa051fc1c61fpermsread"
| sha1sum
1a947ced7375aadf970ccd7ee2bb792b81d5ed1c
```

# Hash length extension attack

- Construct new request using *hashpump*[1]
- Takes signature (i.e., old hash
  - detects hash format
  - supported MD5, SHA1, SHA256, SHA512
- Takes old input data after SECRET
- Takes key length (i.e., length of SECRET)
- Takes message/data to add

- Calculates new padding automatically
- Calculates new valid signature / hash



[1]https://github.com/bwall/HashPump

```
$ ./hashpump
Input Signature:
1a947ced7375aadf970ccd7ee2bb792b81d5ed1c
Input Data: api_key44fefa051fc1c61fpermsread
Input Key Length: 6
Input Data to Add:
api_key44fefa051fc1c61fpermswriteactionevil
f43a9343fd2ee59dc43cfcf32ce52885a85753f8
api_key44fefa051fc1c61fpermsread\x80\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x010api_key44fefa051fc1c61fpermswriteac
tionevil
```

Check if signature is valid with correct SECRET:

```
$ perl -e 'print
"SECRETapi_key44fefa051fc1c61fpermsread\x80\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x010api_key44fefa051fc1c61fperms
writeactionevil"' | sha1sum
f43a9343fd2ee59dc43cfcf32ce52885a85753f8
```

# Hash length extension attack

- Details:
  - For the exploit to be practical they needed to overwrite existing http variables by adding the variable ‘a=’ to the beginning
  - Then concatinated input to hash function would stay the same but application logic does not use unknown variables

```
http://www.flickr.com/services/auth/?
a=api_key=44fefa051fc1c61fpermsread[padding]
&api_key=44fefa051fc1c61f
&perms=write
&action=evil
&api_sig=f43a9343fd2ee59dc43cfcf32ce52885a85753f8
```

# Hash length extension attack

- Solutions:
  - use other hash function not vulnerable to hash length extension attack e.g., SHA3
  - use HMAC as a MAC  which is not vulnerable to length extension attacks by construction
    - roughly constructed like that:

$$HMAC(K, m) = H\Big((K' \oplus opad)||H\big((K' \oplus ipad)||m\big)\Big)$$

# Other examples of Hash function usage / misuse

- Bitcoin/Cryptocurrency Brain Wallet
  - Idea: Derive ECDSA secret and public key from (memorizable) pass phrase e.g., (roughly):
    - str := "*how much wood could a woodchuck chuck if a woodchuck could chuck wood*"
    - secret key d := SHA256(str)
    - public key pk := dG
      - either compressed (x) or uncompressed (x,y) form
    - pkhash := RIPEMD160( SHA256( pk ) )
    - address := Base58( pkhash )
      - e.g.,: 18s9tQHgcyykbqJkuHaxUad9vrWSXf9aqC

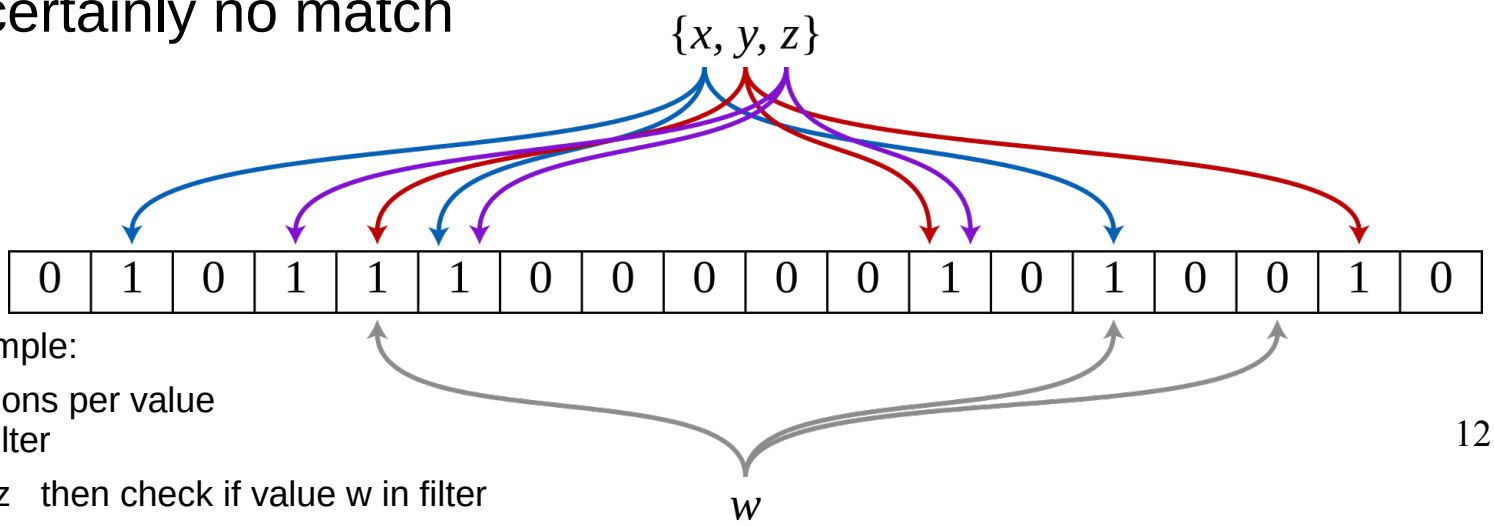
# Other examples of Hash function usage / misuse

- Not a good idea
  - Blockchain = big (unsalted) “password” hash database!
- The Bitcoin Brain Drain [1]
- Brainflayer [2]
  - 130 K guesses/second against entire blockchain (on laptop)
  - GPU/FPGA/ASIC acceleration possible

[1] https://allquantor.at/blockchainbib/#vasek2016bitcoin
[2] https://github.com/ryancdotorg/brainflayer

# Bitcoin brain wallet attack with bloom filters and brute force

- Idea of the attack to be efficient:
  - ~ 80,000,000 BTC addresses to one 512 MB Bloom filter
  - match each address to ~ 20 locations in bitmask
  - difference to other bloom filters, addresses are only bit sliced
  - check against all addresses at once with outputs:
    - probable match => slower check to identify false positives
    - certainly no match




Bloom filter example:

- 3 hash functions per value
  => 3 bits in filter
- 3 values x,y,z then check if value w in filter

# Bitcoin brain wallet attack with bloom filters and brute force

Successfully cracked:

- "*how much wood could a woodchuck chuck if a woodchuck could chuck wood*"
  ~ 250 BTC
- ""
  ~ 50 BTC
- "*Down the Rabbit-Hole*"
  ~ 85 BTC
- "*The Quick Brown Fox Jumped Over The Lazy Dot*"
  ~ 85 BTC

# Bitcoin brain wallet attack with bloom filters and brute force

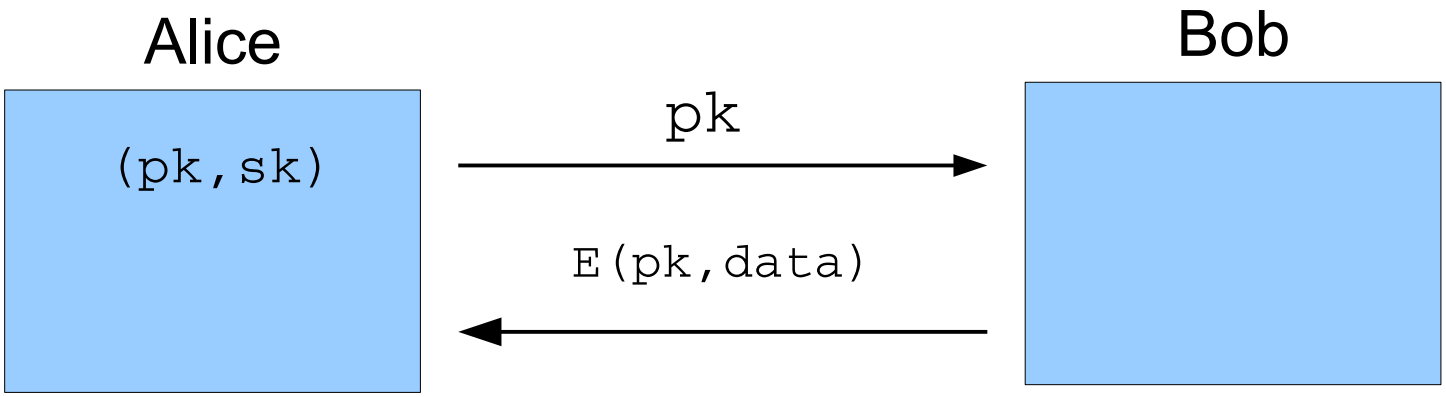
- “gate gate paragate parasamgate bodhi svaha”
- “The Persistence Of Memory”
- “QTC”
- “644122178”
- “8964009”
- “que me lleve la muerte”
- “one two three four five six seven”
- “it’s a secret to everybody”
- “Ph’nglui mglw’nafh Cthulhu R’lyeh wgah’nagl fhtagn”
- “my hovercraft is full of eels”
- “Interior Crocodile Alligator”
- “No need to worry, my accountant handles that”
- “tomb-of-the-unknown-soldier-identification-badge”
- “permit me to issue and control the money of a nation and i care not who makes its laws”
- “who is john galt”
- “Live as if you were to die tomorrow. Learn as if you were to live forever.”

# Asymmetric cryptography (Public-key cryptography)

# Public-key encryption



- High level perspective on public-key encryption only
- Alice generates public private/secret key pair `(pk, sk)`
- Alice transmits `pk` to bob



# Public-key encryption



**Def:** a public-key encryption system is a triple of algs. $(G, E, D)$

- $G()$: **Generation** algorithm that outputs a key pair $(pk, sk)$
- $E(pk, m)$: **Encryption** algorithm that takes a message $m \in M$ and outputs a ciphertext $c \in C$
- $D(sk, c)$: **Decryption** algorithm that takes a ciphertext $c \in C$ and outputs a message $m \in M$ or $\perp$

**Consistency:** $\forall(pk, sk)$ that are output of $G()$:

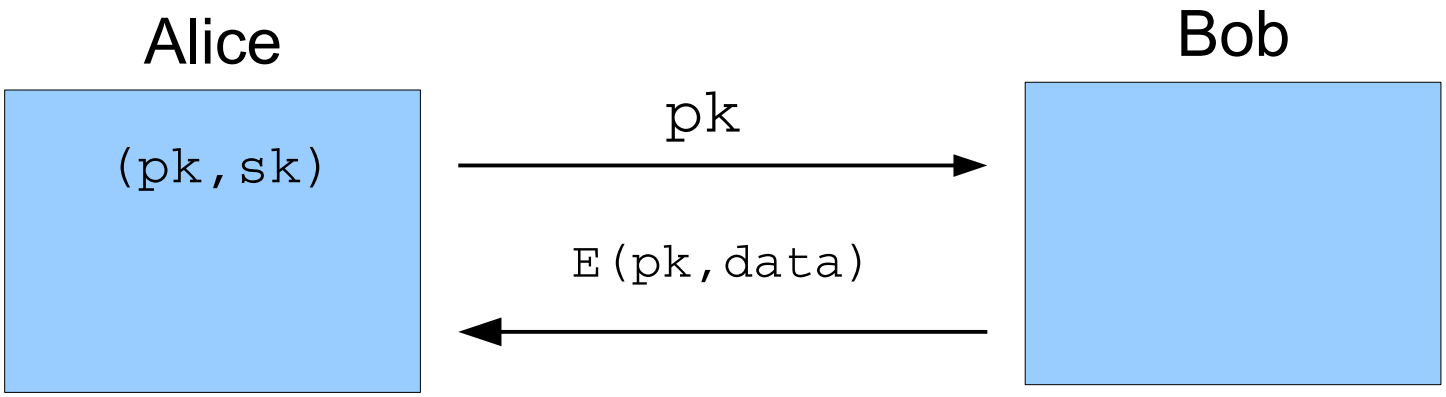
$$\forall m \in M : D(sk,\ E(pk, m)\ ) = m$$

Note: $M$ and $C$ are defined as the message and the key space containing every possible message/ciphertext.

# Public-key encryption



- High level perspective on public-key encryption only
- Alice generates public private/secret key pair `(pk, sk)`
- Alice transmits `pk` to bob

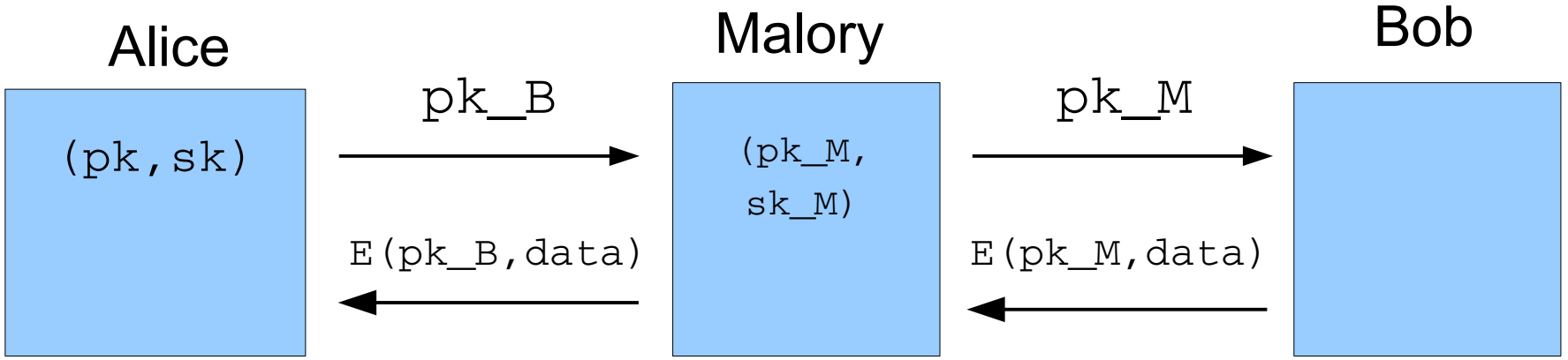



What is a potential security issues here?

# Public-key encryption



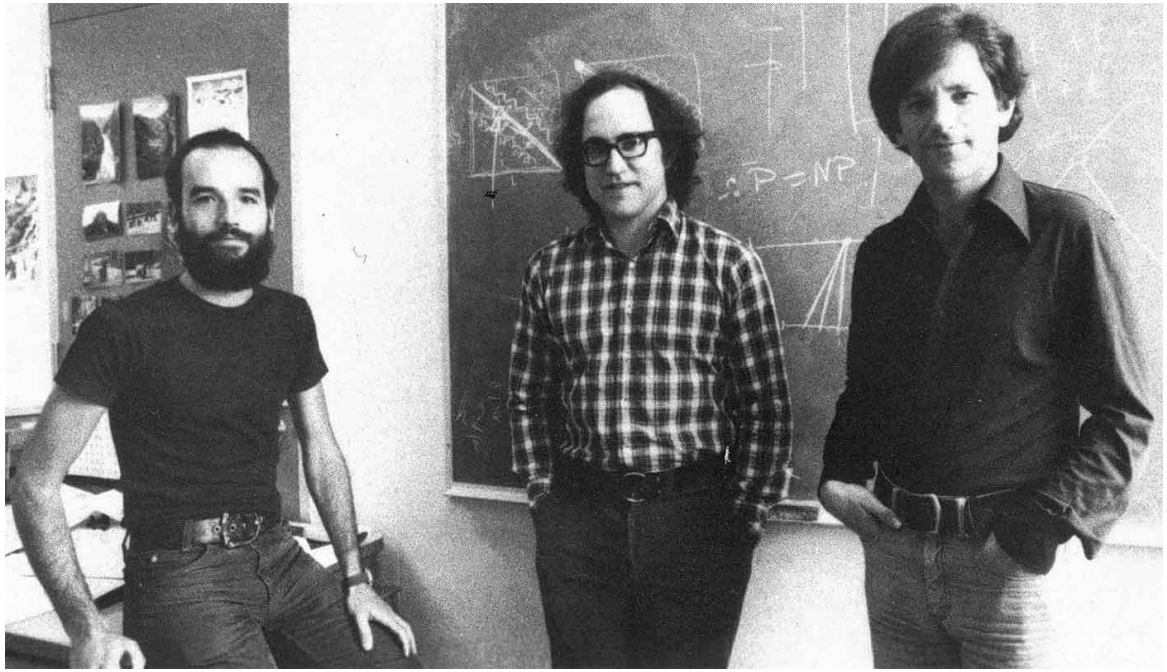
- Vulnerable to Man-in-the-middel (MitM) attack by *Malory*
- Therefore key authenticity is important!
  - This problem is **not solved** with asymmetric cryptography
  - In Practice trust anchor is required: **PKI** (Public Key Infrastructure) (HTTPS), Web of trust (GPG), business card with fingerprint, …



# RSA



- **R**ivest **S**hamir **A**delman (RSA)
  - Adi Shamir, Ron Rivest, Leonard Adelman (all MIT at that time):
- Algorithm first published 1977
- Key length of > 2048 bits still considered secure [2] if correctly implemented




[1] https://claudiodinardo.com/content/images/2017/08/shamir-rivest-adleman.jpg

[2] https://www.keylength.com/

# Trapdoor function



**Def:** Trapdoor function is a triple of efficient algs. $(G, F, F^{-1})$

- $G()$ : Generation algorithm ouptus $(pk, sk)$
- $F(pk, \cdot)$: The $pk$ defines a mapping function $X \to Y$
- $F^{-1}(sk, \cdot)$: Defines a function $Y \to X$ that **inverts** $F(pk, \cdot)$

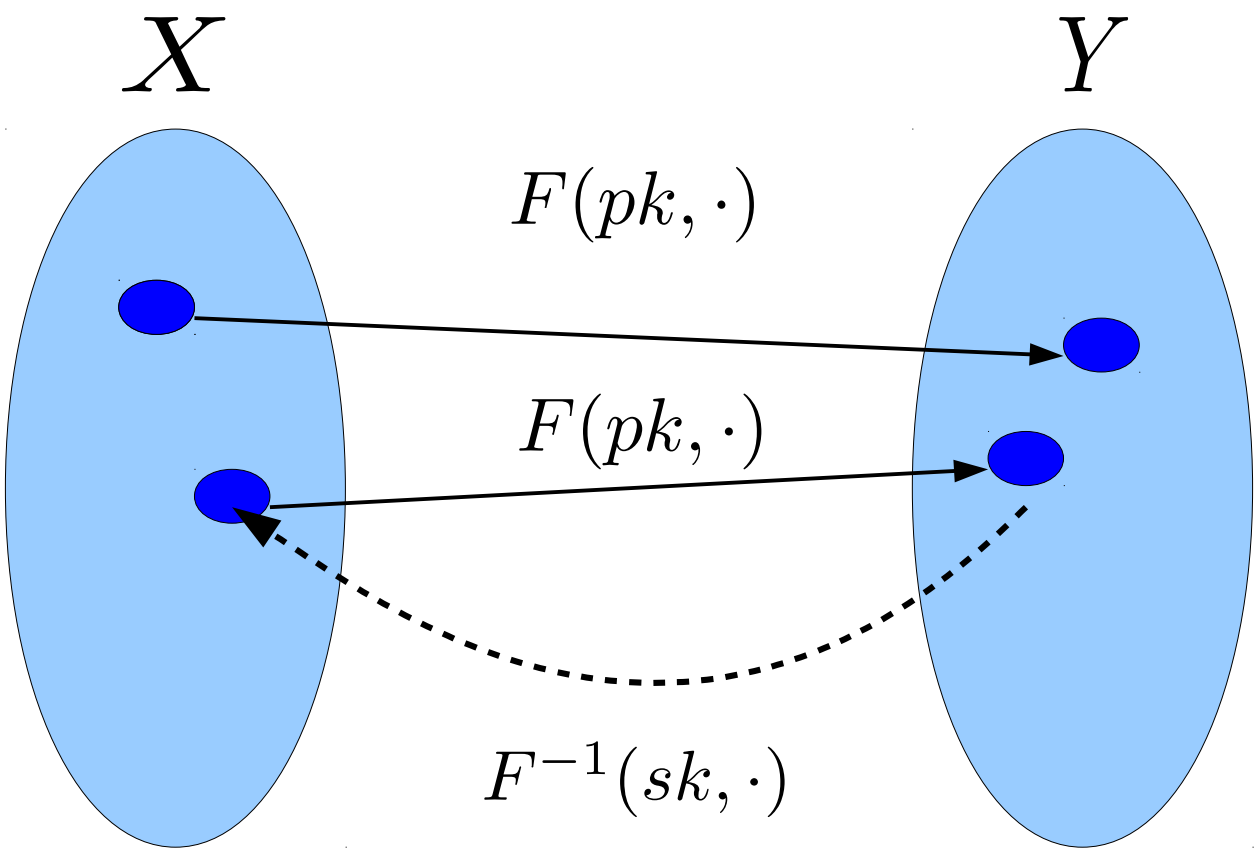
It holds that $\forall(pk, sk)$ that are output of $G()$:

$$\forall x \in X : F^{-1}(sk,\ F(pk, x)\ ) = x$$

Note: If the set $X = Y$ then it is called a trapdoor *permutation*.
A **secure** trapdoor function/permutation $F(pk, \cdot)$ is one-way without the trapdoor $sk$

# Trapdoor function





# Number theory (quick reference)



- $N$ is a **composite number** i.e., the product of two primes $N = pq$
- The set $\mathbb{Z}_N$ is the set modulo $N$ i.e., $\mathbb{Z}_N = \{0, 1, 2, ..., N-1\}$
- The set $\mathbb{Z}_N^*$ is the set of **invertible elements** in $\mathbb{Z}_N$. (We are only interested in *multiplicative* inverses here.)
- An element $x \in \mathbb{Z}_N$ is invertible iff $gcd(x, N) = 1$ i.e., there exists an element $x^{-1}$ s.t. $x \cdot x^{-1} = 1 \pmod{\mathbb{Z}_N}$ Examples:
  - In $\mathbb{R}$ the inverse of 3 would be $1/3$ since $3 \cdot 1/3 = 1$
  - In $\mathbb{Z}_\mathbb{N}$ the inverse of 2 would be $\frac{N+1}{2}$ since $2 \cdot \frac{N+1}{2} = N + 1 = 1 \pmod{\mathbb{Z}_N}$
- The number of invertible elements $\mathbb{Z}_N^*$ in $\mathbb{Z}_N$ is given by $\varphi(N) = (p-1)(q-1) = N - p - q + 1 \approx N - 2\sqrt{N} \approx N$

# Number theory (quick reference)



**Eulers theorem:** $\forall x \in \mathbb{Z}_N^* : x^{\varphi(N)} = 1 \pmod{\mathbb{Z}_N}$
Example:

- $\mathbb{Z}_{15} = \{0, 1, 2, ..., 14\}$ where $N = pq = 3 \cdot 5$
- $\varphi(15) = |\mathbb{Z}_{15}^*| = |\{1, 2, 4, 7, 8, 11, 13, 14\}| = 8$
- Since $p, q \in Prime$ this can be calculated by $(p-1)(q-1) = 2 \cdot 4 = 8$
- $4 \in \mathbb{Z}_{15}^*$
- $4^{\varphi(15)} = 4^8 = 32 = 1 \pmod{\mathbb{Z}_{15}}$

# Number theory in Sage Math



```
sage: Zn = IntegerModRing(15); Zn # or just Integers(15)
Ring of integers modulo 15
sage: Zn.list() # list of all elements
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
Sage: 6*3 # calculation performed in Z
18
sage: Z = Integers(); Z
Integer Ring
sage: 6*3 == Z(6*3)== ZZ(6*3) # implicit or other name for Z
True
sage: mod(6*3,15)
3
sage: Zn(6*3)
3
```

# Number theory in Sage Math



```
sage: Zn.list()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
Sage: Zn.order() # number of elements
15
sage: Zn_inv = [ x for x in Zn if Zn(x**8) == Zn(1) ]; Zn_inv
[1, 2, 4, 7, 8, 11, 13, 14] # invertible elements
sage: Zn(4**8)
1
sage: mod(4**8,15)
1

sage:inverse_mod(7,15)   # get the inverse of element
13
sage: Zn(7*13)
1
```

# RSA trapdoor



**Def:** The **RSA** trapdoor permutation is a triple of efficient algs. $(G, F, F^{-1})$

- $G()$ : 1) choose random primes $p, q \approx 1024$ bits each and compute the 2048 bit RSA ***modulus*** $N = pq$
  2) choose integers $e, d$ such that $e \cdot d = 1 \pmod{\varphi(N)}$
  3) output $pk = (N, e)$ and $sk = (N, d)$
- $F(pk, x) = x^e \pmod{\mathbb{Z}_N}$
  This is a function mapping all invertable elements $\mathbb{Z}_N^* \rightarrow \mathbb{Z}_N^*$
- $F^{-1}(sk, y) = y^d \pmod{\mathbb{Z}_N}$

To show that $F^{-1}(sk,\ F(pk, x)\ ) = x$ we use the fact that there exists some $k$ such that $ed = k \cdot \varphi(N) + 1$ since $e \cdot d = 1 \pmod{\varphi(N)}$

$$y^d = (x^e)^d = x^{ed} = x^{k \cdot \varphi(N)+1} = \left(x^{\varphi(N)}\right)^k \cdot x = (1)^k \cdot x = x$$

Note: $e$ is called the ***encryption exponent***, and $d$ the ***decryption exponent***

# RSA security



- The difficulty of computing $\varphi(N)$, without knowing the factorization of $N$, is the difficulty of computing $d$.
- This is refered to as the *RSA problem*, which can be solved by factoring $N$ and computing $\varphi(N)$.
- Knowing $\varphi(N)$, the multiplicative inverse of $e$ an be calculated fast.

$$\varphi(N) = (p-1)(q-1)$$
$$e \cdot e^{-1} = 1 \pmod{\varphi N}$$
$$e^{-1} = d$$

# Textbook/raw RSA example (insecure)



- Lets pick some primes $p = 2$ and $q = 11$ and compute $N = pq = 22$
- Compute $\varphi(N) = (p-1)(q-1) = 10$
- Find some $e$ such that:
  - It is *relatively prime* to $\varphi(N)$ i.e., $gcd(e, \varphi(N)) = 1$
  - There is a $d$ such that $e \cdot d = 1 \pmod{\varphi(N)}$
- For our example set $e = 3$ and $d = 7$ i.e., $3 * 7 = 1 \pmod{\varphi(N)}$
- Publish $\boldsymbol{pk = (N, e) = (22, 3)}$ and store $\boldsymbol{sk = (N, d) = (22, 7)}$.
- **Encryption:** of message $m = 8$
  - $c = m^e \pmod{N} = 8^3 \pmod{2}2 = 512 \pmod{2}2 = 6$
- **Decryption:** of ciphertext $c = 6$
  - $m = c^d \pmod{N} = 6^7 \pmod{2}2 = 279936 \pmod{2}2 = 8$

# Textbook/raw RSA example (insecure)



```
#!/usr/bin/sage
def rsa(size=2^1024,e=None):
  proof = (size <= 2^1024) # turn off for large values
  p = random_prime(size, proof=proof)
  q = random_prime(size, proof=proof)
  if p==q: return None
  n = p * q
  phi_n = (p-1) * (q-1)
  if not e:
    while True:
        e = ZZ.random_element(1,phi_n)
        if gcd(e,phi_n) == 1: break
  d = inverse_mod(e,phi_n)
  print "(Modulus, enc. exponent, dec. exponent) = ",(n,e,d)
  return n,e,d
```

[1] https://www.math.ucdavis.edu/~anne/FQ2010/Number_Theory_RSA.html

# Textbook/raw RSA example (insecure)



```
# …
def rsa_enc(n,e,m):
  return pow(m,e,n)

def rsa_dec(n,d,c):
  return pow(c,d,n)
```

- Load the code file in sage
- Then functions can be used in this sage session

```
$ sage
sage: load("./rsa.sage")

sage: n,e,d = rsa(2**5) # params
(Modulus, enc. exponent, dec.
exponent) =
(22, 3, 7)

sage: rsa_enc(n,e,8) # encryption
6
sage: rsa_dec(n,d,6) # decryption
8
```

# Textbook/raw RSA example (insecure)



```
# …
def rsa_enc(n,e,m):
  return pow(m,e,n)

def rsa_dec(n,d,c):
  return pow(c,d,n)
```

- Load the code file in sage
- Then functions can be used in this sage session

```
$ sage
sage: load("./rsa.sage")

sage: n,e,d = rsa(2**5) # params
(Modulus, enc. exponent, dec.
exponent) =
(22, 3, 7)

sage: rsa_enc(n,e,8) # encryption
6
sage: rsa_dec(n,d,6) # decryption
8
```

Why is this example obviously insecure?

# Textbook/raw RSA example (insecure)



Parameters are too small, N can be factorized !

```
# Alice

$ sage
sage: load("./rsa.sage")

sage: n,e,d = rsa(2**5)
(Modulus, enc. exponent,
dec. exponentn) =
(22, 3, 7)

sage: rsa_enc(n,e,8)
6
sage: rsa_dec(n,d,6)
8
```

```
# Attacker

sage: n = 22, e=3 # pk Alice
sage: factor(n)
2 * 11
sage: phi_n = (2-1)*(11-1); phi_n
10
sage: d = inverse_mod(e,phi_n);d
7
```

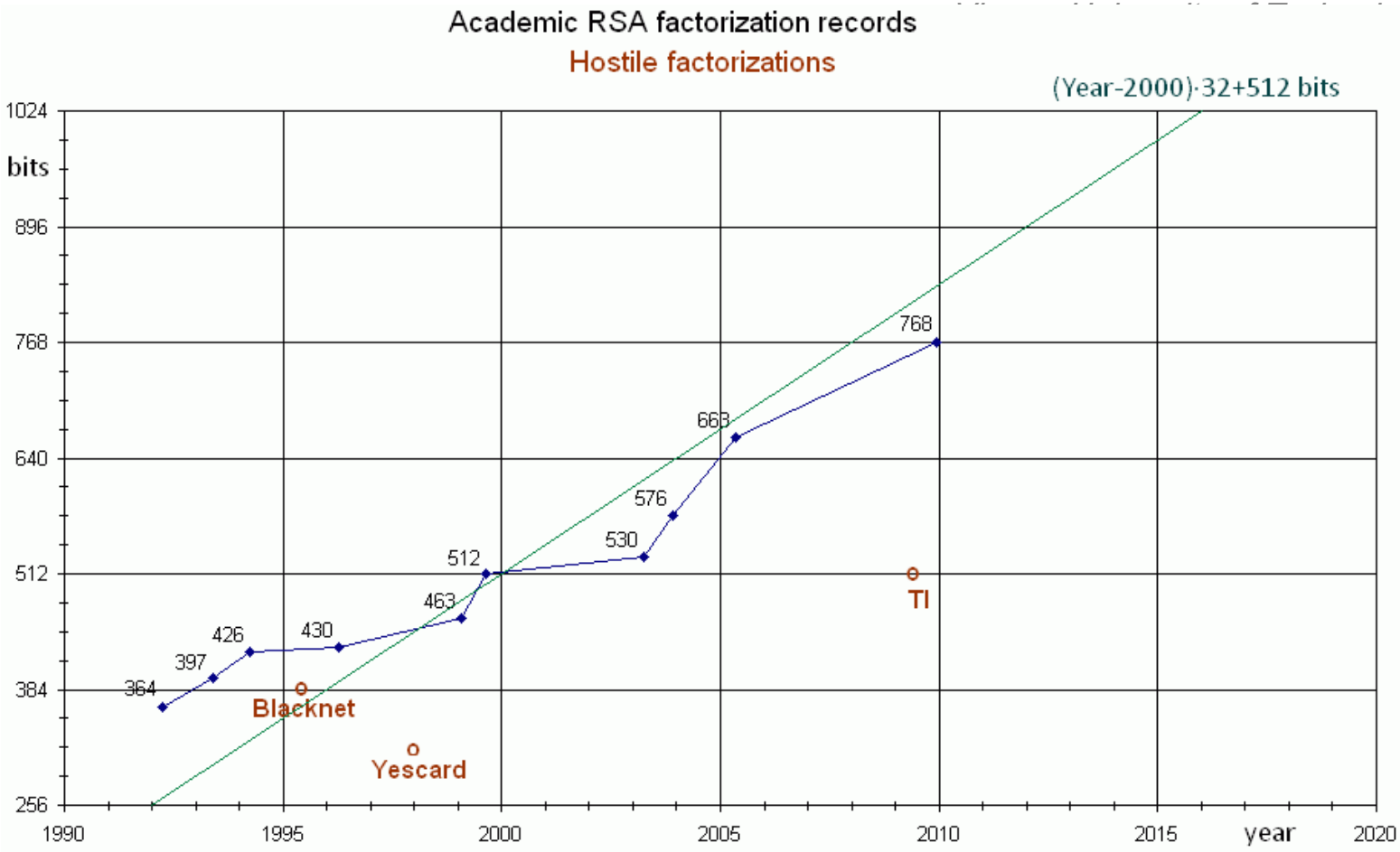
# How fast is factoring?



```
# 64 bit key
sage: time factor(random_prime(2**32)*random_prime(2**32))
CPU times: user 8 ms, sys: 0 ns, total: 8 ms
Wall time: 10.6 ms
# 128 bit key
sage: time factor(random_prime(2**64)*random_prime(2**64))
CPU times: user 76 ms, sys: 0 ns, total: 76 ms
Wall time: 85.3 ms
# 192 bit key
sage: time factor(random_prime(2**96)*random_prime(2**96))
CPU times: user 6.56 s, sys: 12 ms, total: 6.57 s
Wall time: 6.59 s
# 256 bit key
sage: time factor(random_prime(2**128)*random_prime(2**128))
CPU times: user 7min 39s, sys: 656 ms, total: 7min 40s
Wall time: 7min 40s
```

# How fast is factoring?






[1] https://i.stack.imgur.com/VSwml.png

# How fast is factoring?



- Prime factors must be of **approximately the same size.**
- If one prime factor is too small then factoring is easier [1] !

```
sage: time factor(3*random_prime(2**128))
CPU times: user 24 ms, sys: 0 ns, total: 24 ms
Wall time: 23.9 ms
sage: time factor(3*random_prime(2**512))
CPU times: user 644 ms, sys: 0 ns, total: 644 ms
Wall time: 645 ms
sage: time factor(3*random_prime(2**1024))
CPU times: user 8.25 s, sys: 0 ns, total: 8.25 s
Wall time: 8.25 s
sage: time factor(3*random_prime(2**2048))
CPU times: user 1min 55s, sys: 124 ms, total: 1min 55s
Wall time: 1min 55s
```

[1] https://en.wikipedia.org/wiki/Lenstra_elliptic_curve_factorization

# Textbook/raw RSA example (insecure)

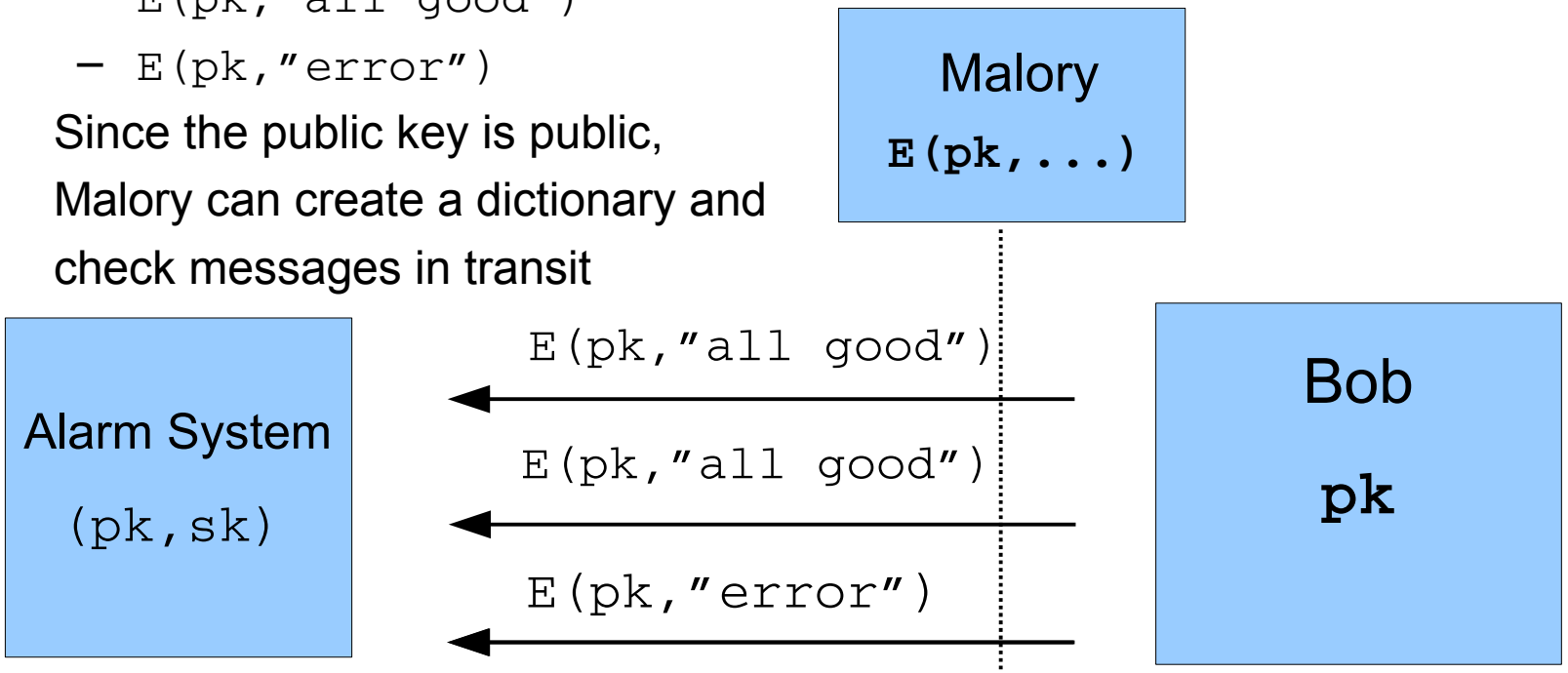


- Lets pick some primes $p$ and $q$ and compute $N = pq$
- Compute $\varphi(N) = (p-1)(q-1)$
- Finde some $e$ such that:
  - It is *relatively prime* to $\varphi(N)$ i.e., $gcd(e, \varphi(N)) = 1$
  - There is a $d$ such that $e \cdot d = 1 \pmod{\varphi(N)}$
- Publish $\boldsymbol{pk = (N, e)}$ and store $\boldsymbol{sk = (N, d)}$.
- **Encryption:** of message $m$
  - $c = m^e \pmod{N}$
- **Decryption:** of ciphertext $c$
  - $m = c^d \pmod{N}$

# Textbook RSA issue: Data patterns visible



- **raw RSA issue:** Deterministic encryption, same plain text leads to same ciphertext. This allows for traffic analysis e.g., alarm system
  - `E(pk,"all good")`
  - `E(pk,"all good")`
  - `E(pk,"error")`
- Since the public key is public, Malory can create a dictionary and check messages in transit



# Textbook RSA issue: Data patterns visible



- To illustrate the problem of determinism and traffic analysis consider:
  - A function to encode a string as number sequence of their 8 bit ASCII codes
  - A function to decode a number into 8 bit sequences of ASCII characters

```
# …
def str_to_num(s,n):
  s = str(s)
  if len(s) > floor(log(n,256)):
      print "Too large for one
rsa round"
      return None
  num = 0
  for i in range(len(s)):
    num += ord( s[i] )*256^i
  return num
```

```
# …
def num_to_str(num):
  num = Integer(num)
  v = []
  while num != 0:
      v.append(chr( num %
256 ))
      num = floor(num/256)
  return ''.join(v)
```

# Textbook RSA issue: Data patterns visible



- Same plain text leads to same cipher text under the same key

```
sage: n,e,d = rsa(2**1024,e=65537)
(Modulus, encryption exponent, decryption exponentn) =
(144130130…)
sage: num = str_to_num("error",n)
sage: c = rsa_enc(n,e,num);c # always the same under (n,e)
286674468414…
sage: m_num = rsa_dec(n,d,c)
sage: m = num_to_str(m_num);m
'error'
```

# Textbook/raw RSA example (insecure)



- Lets pick some primes $p$ and $q$ and compute $N = pq$
- Compute $\varphi(N) = (p-1)(q-1)$
- Finde some $e$ such that:
  - It is *relatively prime* to $\varphi(N)$ i.e., $gcd(e, \varphi(N)) = 1$
  - There is a $d$ such that $e \cdot d = 1 \pmod{\varphi(N)}$
- Publish $\boldsymbol{pk = (N, e)}$ and store $\boldsymbol{sk = (N, d)}$.
- **Encryption:** of message $m$
  - $c = m^e \pmod{N}$
- **Decryption:** of ciphertext $c$
  - $m = c^d \pmod{N}$

# Textbook RSA issue: Integrity not ensured



- The integrity of the ciphertext is not ensured. Therefore, the message/ciphertext is **malleable**.

```
sage: n,e,d = rsa(2**1024,e=65537)
(Modulus, encryption exponent, decryption exponentn) =
(145194...473)
sage: c = rsa_enc(n,e,8); c # small message e.g., sensor value
11983769...072
sage: m = rsa_dec(n,d,c); m
8
sage: c1 = c*c                  # just multiply the value with itself
sage: m = rsa_dec(n,d,c1); m; m==8*8
64
True
sage: c2 = c*rsa_enc(n,e,2) # attacker can calc. factor using pk
sage: m = rsa_dec(n,d,c2); m; m==8*2
16
True
```

# Textbook RSA issue: e th root attack on small m



- eth root attack
- if m is small ($m < n^{1/e}$) and e is small (e.g., e=3) and $m^e << N$
- Then you can simply calculate the eth root within the Integers and decrypt the ciphertext

```
sage: n,e,d = rsa(2**1024,e=3)
(Modulus, encryption exponent, decryption exponentn) =
(756605...4571)
sage: num = str_to_num("A",n); num; hex(num)
65
'41'
sage: c = rsa_enc(n,e=3,65); c
274625

sage: 274625.nth_root(3) # attacker can compute plaintext
65
```

# Textbook/raw RSA (is insecure)



- **Coppersmith attack**
  - Low encryption exponent e and small message m
  - Or partial knowledge of secret key
- **Wiener's attack**
  - Low decryption exponent d
- **Hastad's broadcast attack**
  - Same message to different public keys (moduli) with same (small) e
- **Meet in the middle attack**
  - If small (e.g., 128 bit) non padded)value is transferred that is the product of two (e.g., 64 bit) values.
- **Common factor attacks across multiple keys [2]**
  - Same prime factors in different moduli, Mining your p's and q's
- **Side channels** e.g., *timing*, *power consumption, ...*
- …

[1] https://crypto.stanford.edu/~dabo/papers/RSA-survey.pdf
[2] https://factorable.net/weakkeys12.extended.pdf

# RSA but how?



RSA is just a trapdoor permutation. Not use directly as encryption system (textbook RSA)! It has to be embedded/transformed e.g., OAEP, PKCS1v2.0 or ISO 18033-2.
ISO standard (roughly):

- $(E_s, D_s)$ : Symmetric encryption/decryption system which provides authenticated encryption
- $H() : \mathbb{N} \rightarrow K$ : Secure hash functions that maps elements of $\mathbb{Z}_N$ to secrets keys for the symmetric encryption/decryption system
- $G()$ : Generate RSA params $pk = (N, e)$, $sk = (N, d)$
- $E(pk, m)$ : choose random $x$ in $\mathbb{Z}_N$
  - $y \leftarrow x^e \pmod{\mathbb{Z}_N}$, and $k \leftarrow H(x)$, also **padding** is applied
  - Output and send $\boldsymbol{(y, E_s(k, m))}$
- $D(sk, (y, c)$ : $m = D_s(H(y^d \pmod{\mathbb{Z}_N}), c)$

# (Primer)
# Elliptic curve Cryptography
# (ECC)

# Why ECC?



- Problem(s):
  - Security of RSA relies on hardness of **integer factorization**
  - Security of Diffie-Hellman relies on hardness of **discrete logarithm problem (DLOG)**

- RSA and DH require much higher computational power using longer keys, not only for breaking, but also to compute the ciphertext

- As computational power is also growing and factorization of shorter keys (1024 <=) is already a threat

# Elliptic Curve Cryptography



- ECC also relies on the discrete logarithm problem but over the algebraic structure of *elliptic curves over finite fields*, which makes the (same) problem harder.
  - **Elliptic Curve Discrete Logarithm Problem (ECDLP)**

- Shorter key length for equivalent computational security
  - Means faster computation of ciphertext while retaining hardness against attacks

- Algorithms:
  - Ellipic Curve Diffi-Hellman (ECDH)
  - Elliptic Curve Digital Signature Algorithm (ECDSA)
  - ...

# Elliptic Curve Cryptography



- Key sizes for comparable levels of security [1]:

| Method | Date | Symmetric | Factoring Modulus | Discrete Logarithm Key | Discrete Logarithm Group | Elliptic Curve |
|---|---|---|---|---|---|---|
| [1] Lenstra / Verheul | 2084 | 135 | 7813 6816 | 241 | 7813 | 257 |
| [2] Lenstra Updated | 2090 | 128 | 4440 6974 | 256 | 4440 | 256 |
| [3] ECRYPT II | 2031 - 2040 | 128 | 3248 | 256 | 3248 | 256 |
| [4] NIST | 2016 - 2030 & beyond | 128 | 3072 | 256 | 3072 | 256 |
| [5] ANSSI | 2021 - 2030 | 128 | 2048 | 200 | 2048 | 256 |
| [6] IAD-NSA | - | 256 | 3072 | - | - | 384 |
| [7] RFC3766 | - | 136 | 3707 | 272 | 3707 | 257 |
| [8] BSI | > 2022 | 128 | 3000 | 250 | 3000 | 250 |

[1] https://www.keylength.com/en/compare/

# Elliptic Curve Cryptography



- Comparison of computationally equivalent key sizes for the currently known best effort attacks (bit):

| Symmetric | ECC | RSA/DH/DSA |
|---|---|---|
| 80 | 163 | 1024 |
| 128 | 283 | 3072 |
| 192 | 409 | 7680 |
| 256 | 571 | 15360 |

Performance Analysis of Elliptic Curve Cryptography for SSL – Gupta et al.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.93.8797&rep=rep1&type=pdf

# Elliptic Curve over $\mathbb{F}_p$



**Def.:** Elliptic curve over $\mathbb{F}_p$ requires the following properties:

- Let $\mathbb{F}_p$ prime finite filed
- Let $a, b \in \mathbb{F}_p$ satisfy $4a^3 + 27^b 2 \not\equiv 0$

Then an elliptic curve $E(\mathbb{F}_p)$ over $\mathbb{F}_p$ is defined by:

- The parameters $a, b \in \mathbb{F}_p$
- The set of solutions i.e., points $P = (x, y)$, for all pairs $x, y \in \mathbb{F}_p$ to the *defining euqation*:

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

- The point at infinity $\infty$ or $\mathcal{O} \in E(\mathbb{F}_p)$ s.t., $P + \mathcal{O} = \mathcal{O} + P = P$

# Elliptic Curve Cryptography

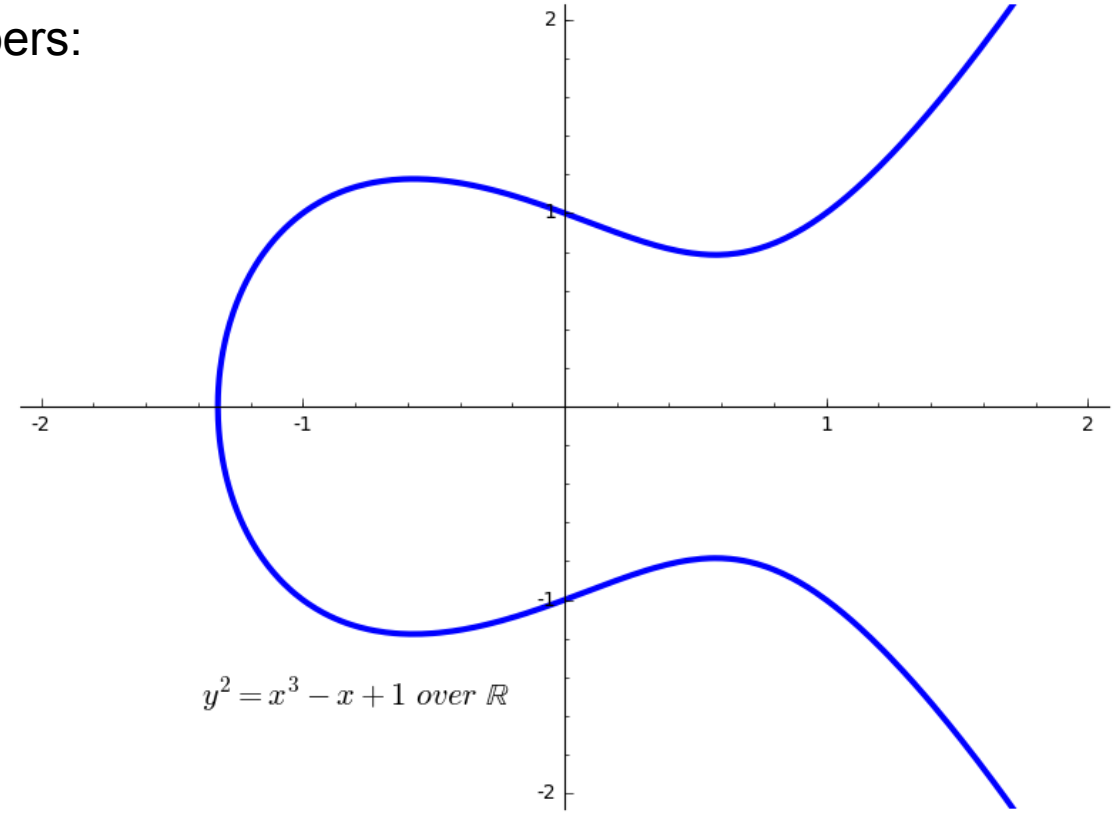


Simplified Weierstraß equation over real numbers:

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

$$a = -1$$

$$b = 1$$



# Elliptic Curve Cryptography



Simplified Weierstraß
equation over real numbers:

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

$$a = -1$$

$$b = 1$$

```
# sage code
EC = EllipticCurve(RR, [0,0,0,-1,1]); EC
p = plot(EC, thickness=3,xmax=2)

p.show(xmin=-2,xmax=2,ymin=-2,ymax=2)
```

# Elliptic Curve Cryptography

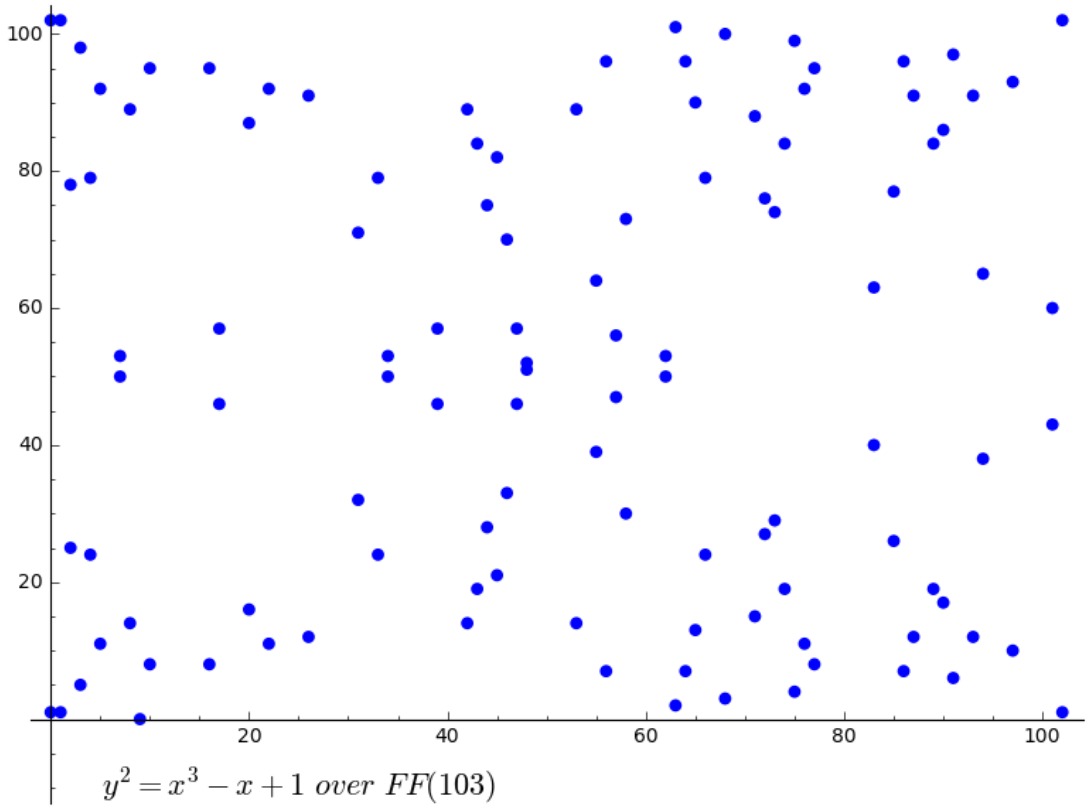


Simplified Weierstraß equation over $\mathbb{Z}_{103}$

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

$$a = -1$$

$$b = 1$$



# Elliptic Curve Cryptography



Simplified Weierstraß equation over $\mathbb{Z}_{103}$

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

$$a = -1$$

$$b = 1$$

```
# sage code
FF = FiniteField(103)
EC = EllipticCurve(FF, [0,0,0,-1,1]); EC

p = plot(EC, size=40)

p.show()
```

# Elliptic Curve Cryptography



- The number of points on $E(\mathbb{F}_p)$ is denoted by $\#E(\mathbb{F}_p)$ and can be calculated by the Hasse Theorem:

$$p + 1 - 2\sqrt{p} \leq \#E(\mathbb{F}_p) \leq p + 1 + 2\sqrt{p}$$

```
sage: FF = FiniteField(103); FF.order()
103
sage: EC = EllipticCurve(FF,[0,0,0,-1,1]); EC
Elliptic Curve defined by y^2 = x^3 + 102*x + 1 over Finite Field of size
103
sage: EC.order(); EC.order() == len(EC.points())
112
True
sage: 103+1-2*(103.sqrt()) <= EC.order() <= 103+1+2*(103.sqrt()) # Hasse
True
```

# Elliptic Curve Cryptography



- A generator $G$ of the curve has the same order as the curve and hence can generate all points on the curve:

$$\{G, G2, G3, ..., G\#E(\mathbb{F}_p)\}$$

```
sage: G = EC.gen(0); G.xy(); G
(74, 84)
(74 : 84 : 1)  # (x : y : 0 iff point at infinity, 1 otherwise)
sage: G.order()
112
sage: (G*1) == G
True
sage: G*2; G*3; G*111; G*112; G*113
(72 : 76 : 1)
(76 : 11 : 1)
(74 : 19 : 1)
(0 : 1 : 0)
(74 : 84 : 1)
```

# Elliptic Curve Cryptography



Inverse:

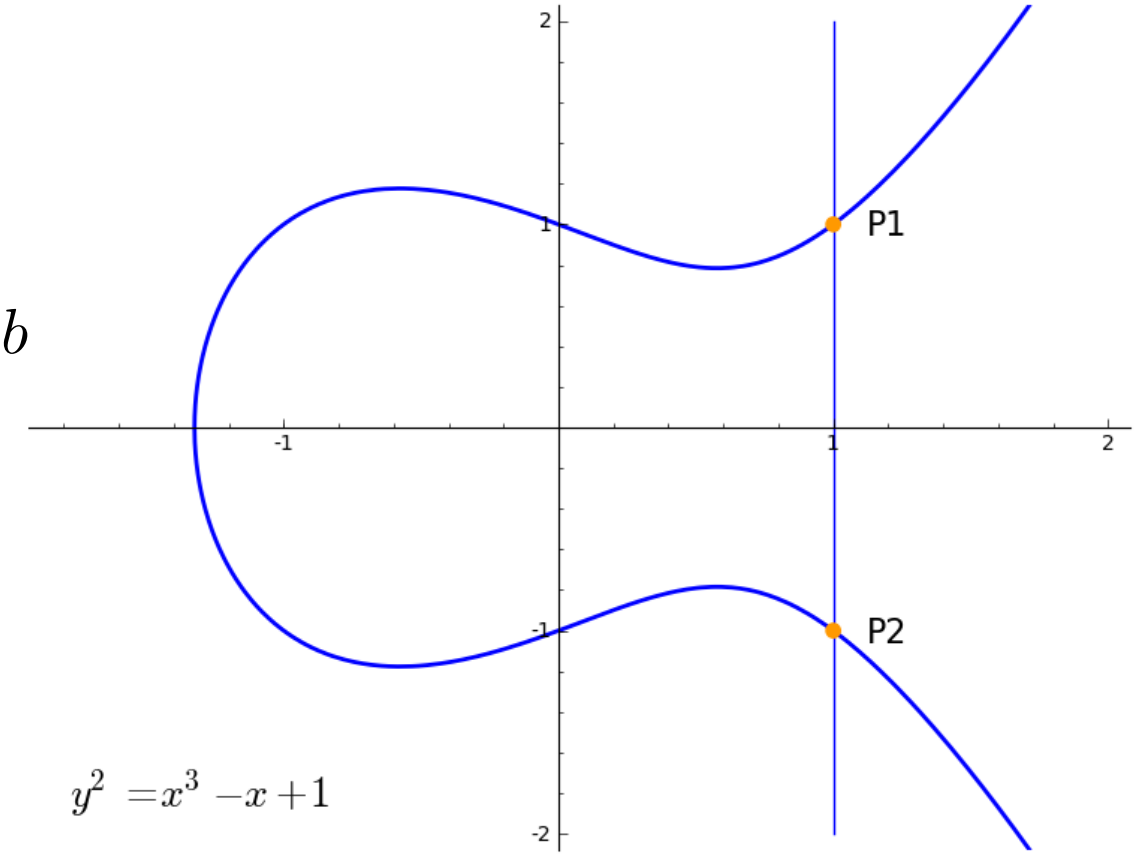
P1=(x, y)=(1,1)

P2=-P1=(x, -y)=(1,-1)

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

$$a = -1$$

$$b = 1$$



# Elliptic Curve Cryptography



Inverse:

P1=(x, y)=(1,1)

P2=-P1=(x, -y)=(1,-1)

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

$$a = -1$$

$$b = 1$$

```
# sage script
EC = EllipticCurve ( RealField (53) , [0 ,0 ,0 , -1 ,1]);
EC
# define two points to add on curve
P1 = EC (1 ,1)
P2 = EC (1 , -1)
# calculate Q
Q = P1 + P2
p = EC . plot ( thickness =2)
p += point ([ P1 . xy () , P2 . xy ()] , size =60 , hue
=0.1 , zorder =4)
p += line ([(P1.xy()[0] , -2) ,(P2.xy()[0] ,2)] ,
rgbcolor =(0 ,0 ,1))
p.show ( xmin = -2 , xmax =2 , ymin = -2 , ymax =2)
```

# Elliptic Curve Cryptography



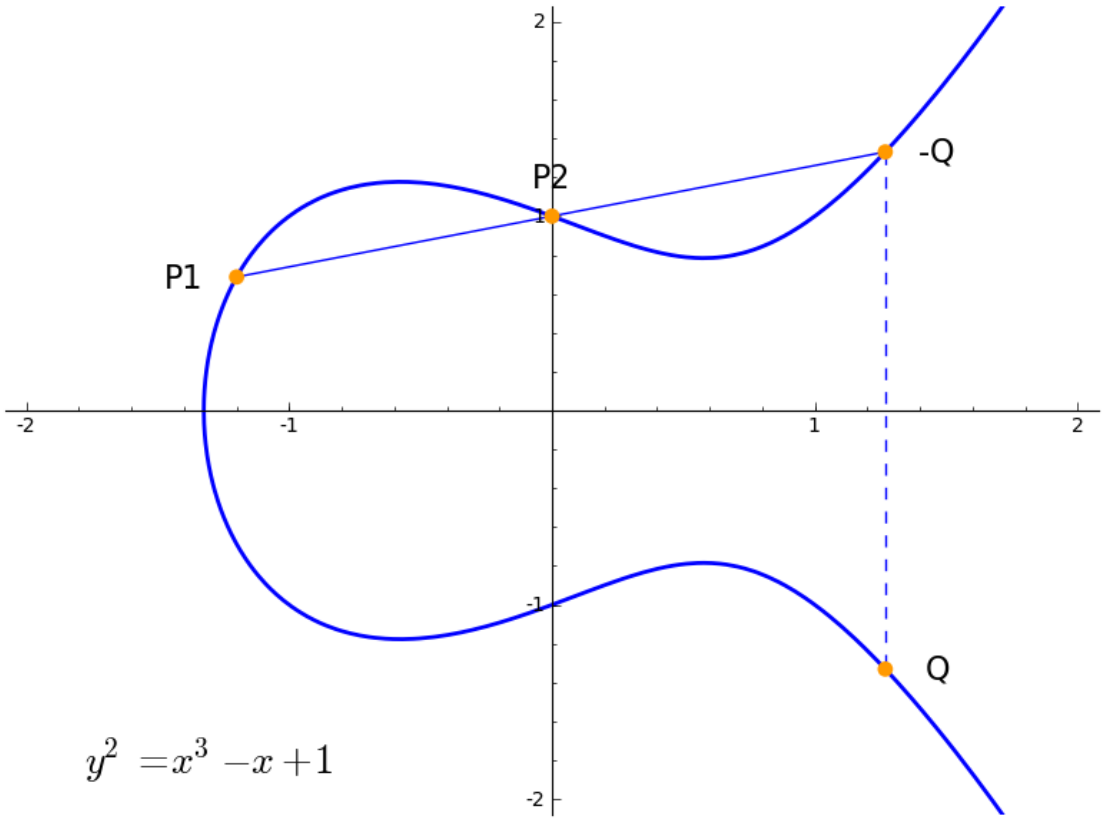
Point addition:

P1+P2=Q

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

$$a = -1$$

$$b = 1$$



# Elliptic Curve Cryptography



Point addition:

P1+P2=Q

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

$$a = -1$$

$$b = 1$$

```
# create elliptic curve over the Reals with 53 bit precision
EC = EllipticCurve ( RealField (53) , [0 ,0 ,0 , -1 ,1]); EC
# define two points to add on curve
P1 = EC ( -1.20 , RealField (54)(1/25* sqrt (5)* sqrt (59)));
P1
P2 = EC (0.0 ,1); P2
# calculate Q
Q = P1 + P2 ; Q
# plot grafics
p = EC . plot ( thickness =2)

p += point ([ P1 . xy () , P2 . xy () , Q . xy () ,
(1.26802422014902 , 1.33071914365621)] , size =60 ,
p += line ([ P1 . xy () ,(1.26802422014902 , 1.33071914365621)]
, rgbcolor =(0 ,0 ,1))
p += line ([(1.26802422014902 , 1.33071914365621) , Q . xy
()] , linestyle =" - -" , rgbcolor =(

p . show ( xmin = -2 , xmax =2 , ymin = -2 , ymax =2)
```

# Elliptic Curve Cryptography



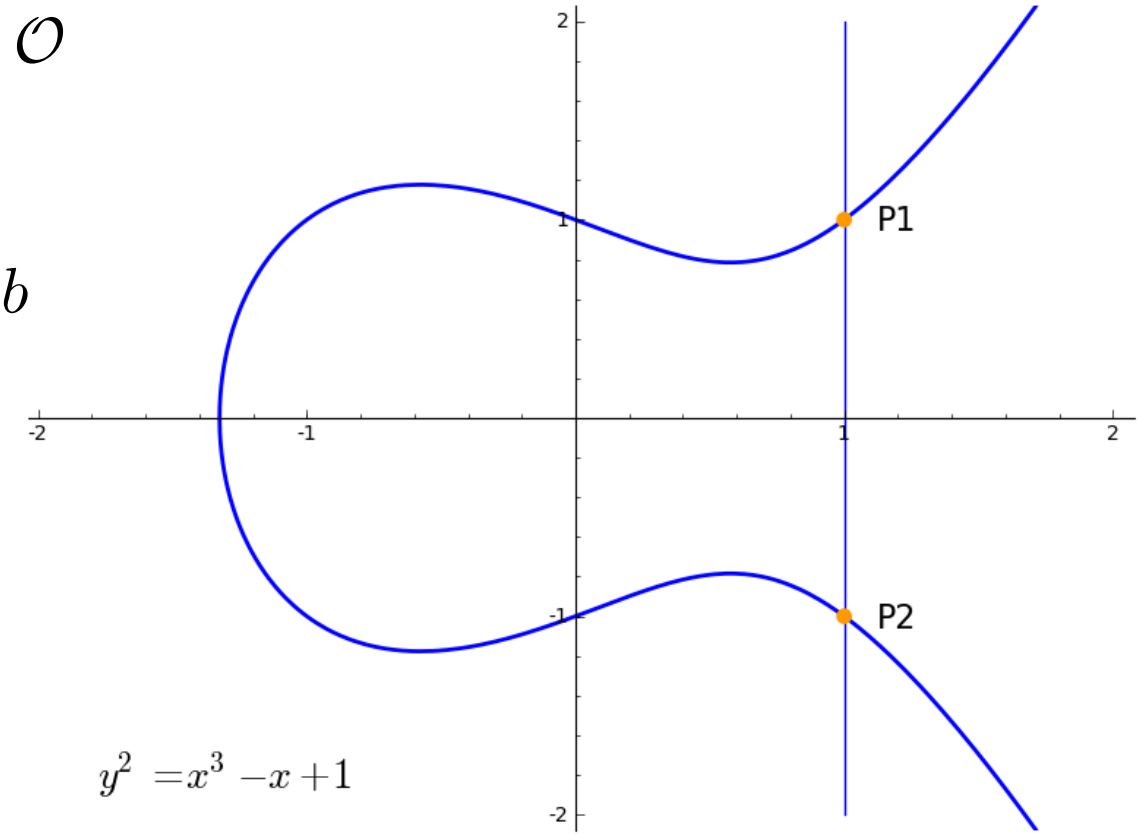
Point addition:

$$P1 + P2 = \infty = \mathcal{O}$$

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

$$a = -1$$

$$b = 1$$



# Elliptic Curve Cryptography



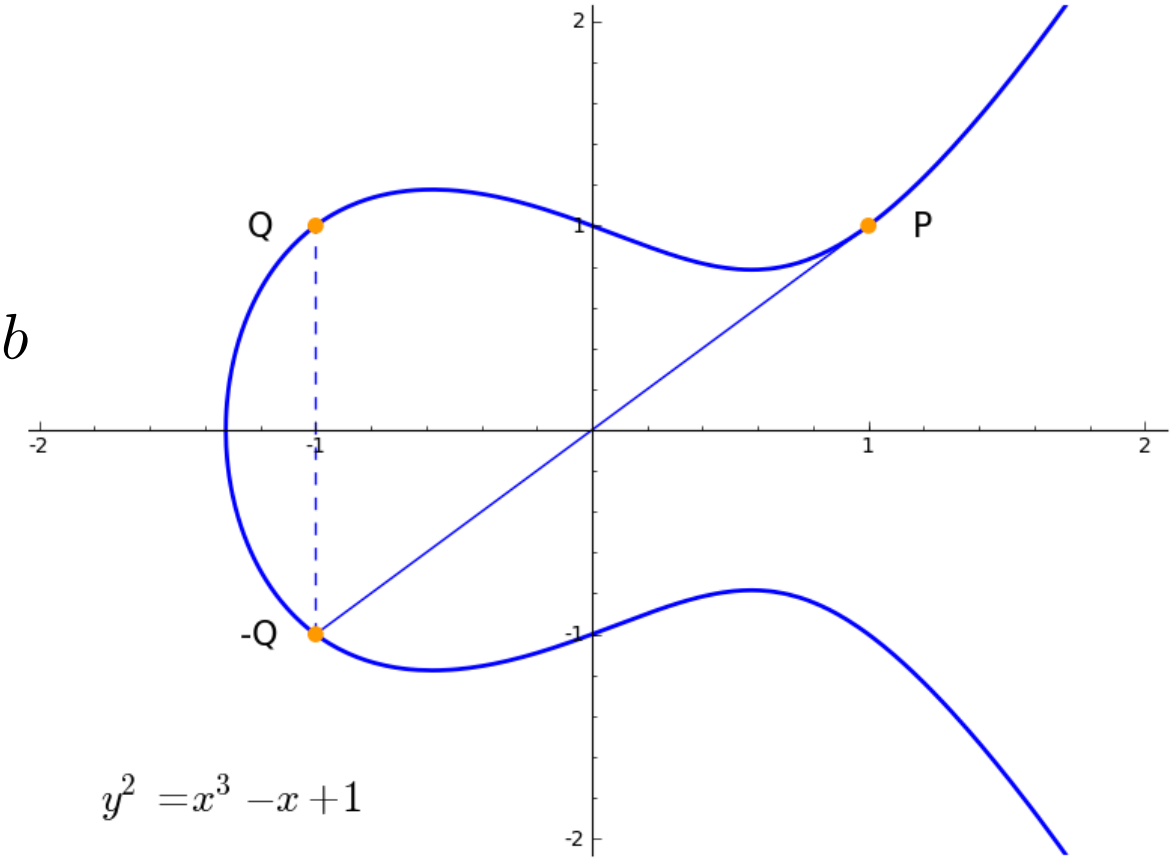
Point multiplying:

P+P = 2P

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

$$a = -1$$

$$b = 1$$



# Elliptic Curve Cryptography



Point multiplying:

P+P = 2P

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

$$a = -1$$

$$b = 1$$

```
# create elliptic curve over the Reals with 53 bit precision
EC = EllipticCurve ( RealField (53) , [0 ,0 ,0 , -1 ,1]); EC
# define two points to add on curve
P = EC (1 ,1); P
# calculate Q
Q = P *2; Q
# plot
p = EC . plot ( thickness =2)
p += point ([ P . xy () , Q . xy () , ( -1.00000000000000 ,
-1.00000000000000)] , size =60 , hue =0.1
p += line ([ P . xy () ,( -1.00000000000000 ,
-1.00000000000000)] , rgbcolor =(0 ,0 ,1))
p += line ([( -1.00000000000000 , -1.00000000000000) , Q . xy
()] , linestyle =" - -" , rgbcolor
p += text (" P " ,(1.20000000000000 , 1.00000000000000) ,
fontsize =16 , color = ' black ')
p += text (" Q " ,( -1.20000000000000 , 1.00000000000000) ,
fontsize =16 , color = ' black ')
p += text (" - Q " ,( -1.20000000000000 , -1.00000000000000) ,
fontsize =16 , color = ' black ')
p += text (" $y ^2 = x ^3 - x + 1 $ " , ( -1.3 , -1.8) ,
fontsize =20 , color = ' black ')
p . show ( xmin = -2 , xmax =2 , ymin = -2 , ymax =2)
```

# Elliptic Curve Cryptography



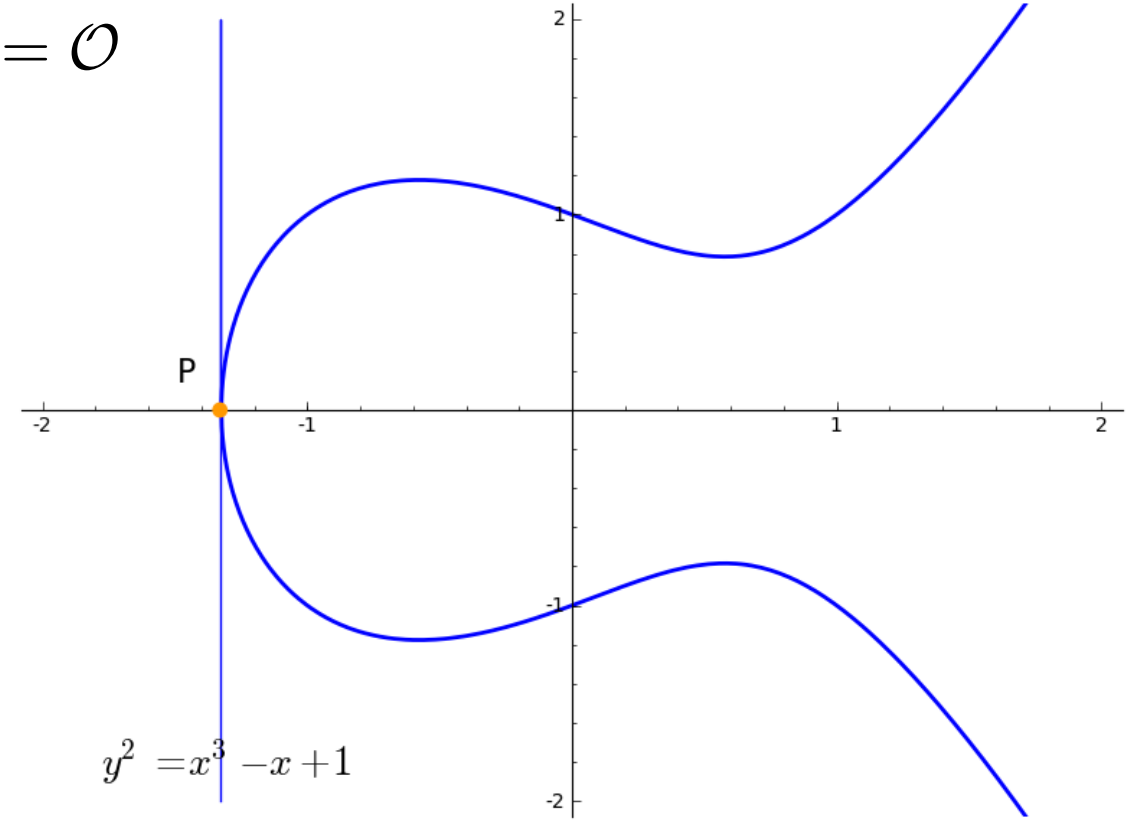
Point multiplying:

$$P + P = 2P = \infty = \mathcal{O}$$

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

$$a = -1$$

$$b = 1$$



# Elliptic Curve Cryptography



In case of ECC over prime fields $\mathbb{F}_p$, where $p \in Prime$,
the following *domain parameters* have to be defined:

- $p$: The prime defining the field $\mathbb{F}_p$, under which the curve operates. All point operations $(+, *)$ are taken moduo $p$.
- $a, b$: Two integers which are the coefficients defining the curve $E()$.
- $G$: The generator- or base-point. Used as a starting point for multiplications.
- $n$: The order of $G$, which is the number of distinct points on the curve which can be computed by multiplying $G$ with a scalar value.
- $\#E(\mathbb{F}_p)$: The number of points on the elliptic curve over $\mathbb{F}_p$
- $h$: The cofactor, i.e. number of points on the elliptic curve divided by $n$

# ECDSA: Signature creation



Given a curve $(p, a, b, G, n)$, and a secret key $d_A$.

- Compute public key $Q_A = d_A * G$
- Compute hash of data $e = hash(m)$
- Generate **random numer** $k$ sucht that $0 < k < n$
- Compute the first part of the signature:
  Calculate point $R = k * G = \begin{pmatrix} r_x \\ r_y \end{pmatrix}$ and form that generate $r = r_x \mod n$, where $r \neq 0$. $k$ and $R$ are ephemeral key pairs
- Then generate the rest of the signature:
  Calculate $s = \frac{e + d_A * r}{k} \mod n$
- Publish the signature $(\boldsymbol{r}, \boldsymbol{s})$ together with the public key $Q_A$

# ECDSA: Signature verification



Given a curve $(p, a, b, G, n)$, a public key $Q_A$ together with a signature $(r, s)$. The signature $(r, s)$ over $m$ is valid iff:

- The signature values are plausible, i.e. $0 < r < n$ and $0 < s < n$.
- Compute the hash of the data $e = hash(m)$
- Invert $s$ modulo $n$, i.e. $w = s^{-1} \bmod n$
- Calculate $u_1 = e * w \bmod n$, and $u_2 = r * w \bmod n$.
- Derive point $P = \begin{pmatrix} p_x \\ p_y \end{pmatrix} = u_1 * G + u_2 * Q_A$
- If $p_x = r \bmod n$, the signature is valid.

# ECDSA: Signature verification



Given a curve $(p, a, b, G, n)$, a public key $Q_A$ together with a signature $(r, s)$. The signature checking algorithm for $(r, s)$ over $m$ is correct because:

$$P = u_1 * G + u_2 * Q_A$$

$$u_1 = e * w \text{ and } u_2 = r * w$$

$$P = e * w * G + r * w * Q_A$$

$$P = e * w * G + r * w * d_A * G$$

$$P = (e * w + r * w * d_A) * G$$

$$P = w * (e + r * d_A) * G$$

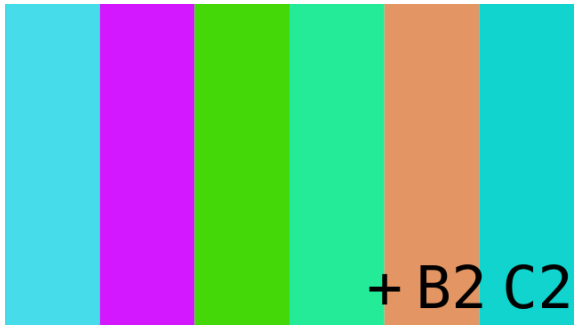
$$w = s^{-1} = \frac{e + d_A * r}{k}^{-1} = \frac{k}{e + d_A * r} \bmod n$$

$$P = \frac{k}{e + d_A * r} * (e + d_A * r) * G = k * G$$

# ECDSA: Signature random reuse



- Sony PS3 ECDSA fail(0verflow)
  - PS3 used code signing to only allow code from trusted sources
  - Nonce was not random
  - Secret key was recovered
  - After fail0verflow presented the attack at 27c3 [1] George Hotz (geohot) released the private key of the PS3 using this techniques together with a “Hello world program” for the PS3
  - Lawsuits followed [2,3]
  - Now the case is settled [2]




[1] https://events.ccc.de/congress/2010/Fahrplan/events/4087.en.html
[2] https://en.wikipedia.org/wiki/Sony_Computer_Entertainment_America,_Inc._v._Hotz
[3] https://en.wikipedia.org/wiki/PlayStation_3_homebrew

# ECDSA: Signature random reuse



Given a curve $(p, a, b, G, n)$, a public key $Q_A$ together with

- a signature $(r, s_1)$ together with a message $m_1$
- a signature $(r, s_2)$ together with a message $m_2$

It can be observed that both signatures used the same value $r$ and hence the same random value $k$.
This can be used to reconstruct the private key $d_A$.

# ECDSA: Signature random reuse



The only unknown variables in this two equations (1) and (2) are, $k$ and $d_A$. To solve for $d_A$ first rearrange the euqations to (3) and (4) and replace $k$ in (5).

$$s_1 = \frac{e_1 + d_A * r}{k} \mod n \tag{1}$$

$$s_2 = \frac{e_2 + d_A * r}{k} \mod n \tag{2}$$

$$k = \frac{e_1 + d_A * r}{s_1} \mod n \tag{3}$$

$$k = \frac{e_2 + d_A * r}{s_2} \mod n \tag{4}$$

$$\frac{e_1 + d_A * r}{s_1} = \frac{e_1 + d_A * r}{s_2} \mod n \tag{5}$$

# ECDSA: Signature random reuse



Then solve for $d_A$.

$$\frac{e_1 + d_A * r}{s_1} = \frac{e_1 + d_A * r}{s_2} \bmod n$$

$$s_2 * (e_1 + d_A * r) = s_1 * (e_2 + d_A * r) \bmod n$$

$$s_2 * e_1 + s_2 * d_A * r = s_1 * e_2 + s_1 * d_A * r \bmod n$$

$$s_2 * e_1 - s_1 * e_2 = s_1 * d_A * r - s_2 * d_A * r \bmod n$$

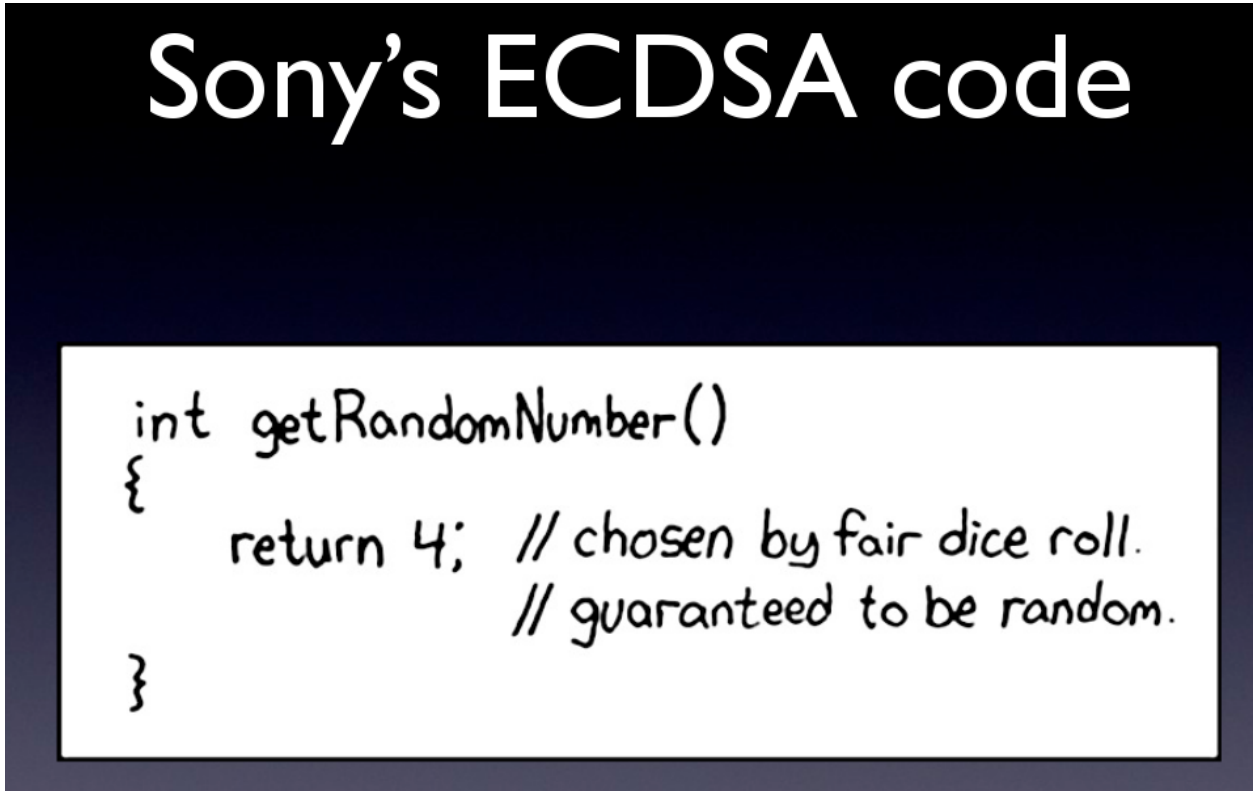
$$s_2 * e_1 - s_1 * e_2 = d_A * r * (s_1 - s_2) \bmod n$$

$$d_A = \frac{s_2 * e_1 - s_1 * e_2}{r * (s_1 - s_2)} \bmod n$$

# ECDSA: Signature random reuse



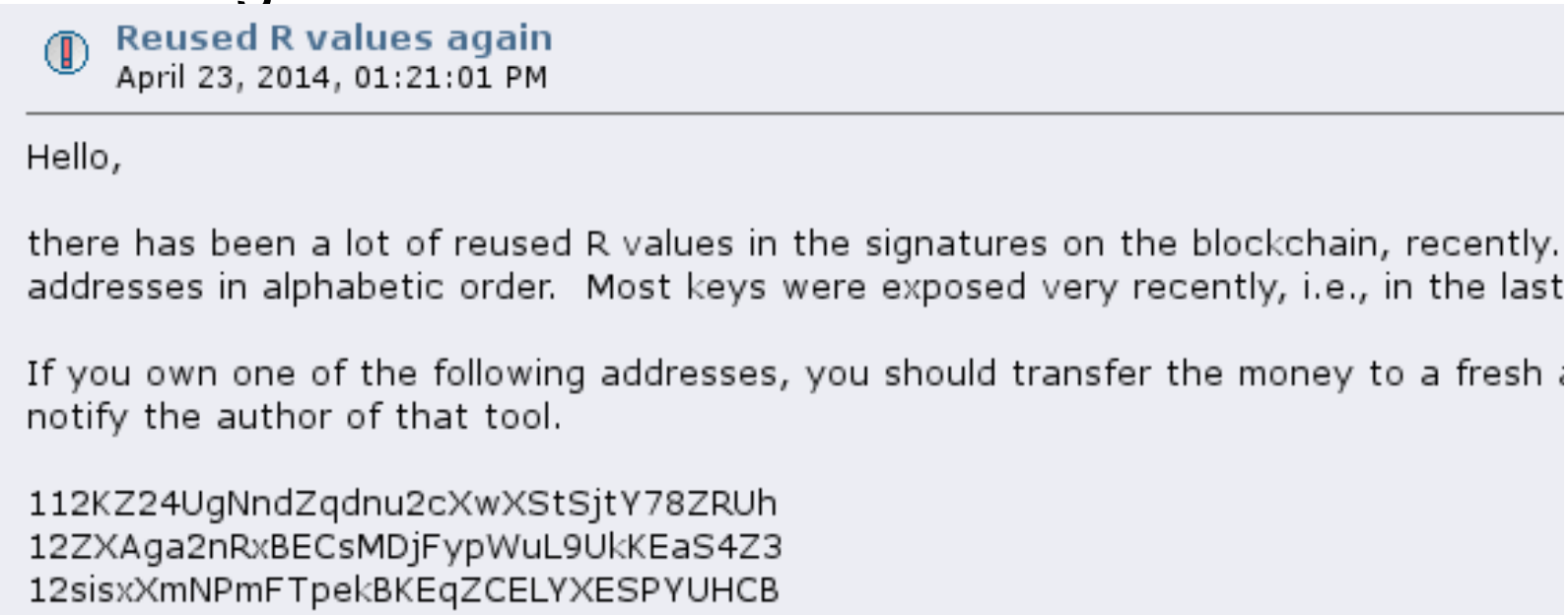
- Sony PS3 ECDSA fail(0verflow)



# Other example: Bitcoin Wallets Online/web wallet Counterwallet.io (Apr,2014)



- Used the same nonce for multiple signed messages



**Reused R values again**
April 23, 2014, 01:21:01 PM

Hello,

there has been a lot of reused R values in the signatures on the blockchain, recently.
addresses in alphabetic order. Most keys were exposed very recently, i.e., in the last

If you own one of the following addresses, you should transfer the money to a fresh
notify the author of that tool.

112KZ24UgNndZqdnu2cXwXStSjtY78ZRUh
12ZXAga2nRxBECsMDjFypWuL9UkKEaS4Z3
12sisxXmNPmFTpekBKEqZCELYXESPYUHCB

[1] https://bitcointalk.org/index.php?topic=581411.0
[2] https://bitcointalk.org/index.php?topic=395761.msg6354587#msg6354587

# ECC Conclusion



- Do **not reuse nonces** in ECDSA!
  - nonce = **n**umber **on**ly use on**ce**
  - Either derive from secure RNG or deterministically
    - http://tools.ietf.org/html/rfc6979

- Important to use **safe** curve and domain parameters for ECC
  - https://safecurves.cr.yp.to/
  - EdDSA
    - https://tools.ietf.org/html/rfc8032
    - https://ed25519.cr.yp.to/
  - Brainpool curves (non-NIST)
    - http://www.ecc-brainpool.org/download/Domain-parameters.pdf
  - NIST curves
    - http://csrc.nist.gov/publications/fips/archive/fips186-2/fips186-2.pdf

- Good online resources on ECC and ECDSA:
  - http://www.secg.org/sec1-v2.pdf
  - http://www.johannes-bauer.com/compsci/ecc/
  - http://www.infosecwriters.com/Papers/Anoopms_ECC.pdf

# References



- Excellent coursera course on cryptography by Dan Boneh
  - https://www.coursera.org/learn/crypto
  - Read-through book for deeper understanding:
  - https://crypto.stanford.edu/~dabo/cryptobook/
  - "A Graduate Course in Applied Cryptography"
    *Dan Boneh and Victor Shoup*
- Handbook and quick reference:
  - "Handbook of Applied Cryptography"
    *Alfred J. Menzes, Paul C. van Oorschot and Scott A. Vanstone*
  - http://cacr.uwaterloo.ca/hac/
- Getting started on cryptographic engineering
  - "Cryptographic Engineering: Design Principles and Practical Applications"
    *Niels Ferguson, Bruce Schneier and Tadayoshi Kohno*
  - Only scratches the surface on some topics, no ECC