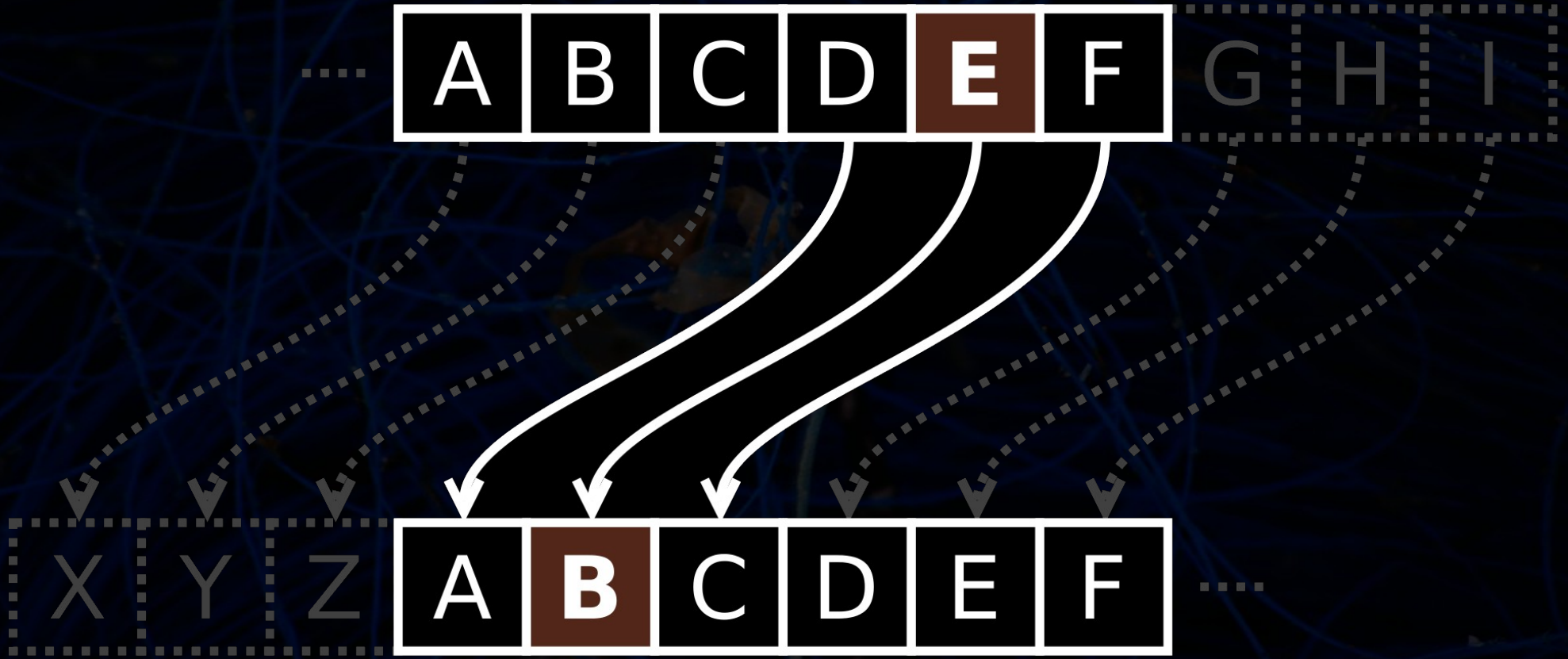
Cryptography



https://commons.wikimedia.org/wiki/File:Caesar_shift_cryptography.png

Lord BugBlue

Mendel Mobach

Does weird things with computers

Crypto & Security Architect

Hack42 Boardmember

WOB / WOO fanatic

Likes to talk about weird interfaces & connections

flickr.com/dvanzuijlekom

What is Cryptography

Secure communication in the presence of adversarial behavior.

Unreadable for interceptors

Verifiable documents

Why Cryptography

Privacy – nobody else is getting wiser

Security – nobody else can impersonate

Both are also false

- Revoking a signature is impossible
- Trusting only on cryptography is stupid

Cryptography basis

- Symmetrical
  - Both parties know the same secret
  - Keep the secret a secret
  - Fast
- A-Symmetrical
  - Public / Private keys (your webserver certificate)
- Hashing
  - From data to hash is easy
  - From hash to data is hard (should be)

Cryptography basis

Do not invent crypto yourself!

Do not write your own crypto!

Use libraries and well used software

If in doubt: use libsodium (will explain later)

Hashing

Works one way (data => hash)

One bit changed, output is very different

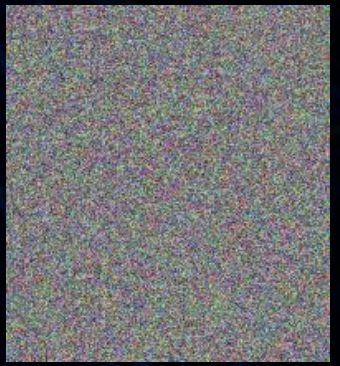
```
mendel@laborans:~> echo "Hallo Wereld" | openssl dgst -sha1
(stdin)= ac04a90c8e7813aa82dcc609378bcbd15b075f02
mendel@laborans:~> echo "Hallo wereld" | openssl dgst -sha1
(stdin)= 31a374196739f3b03d3875085cfdb29b5b3de16f
```

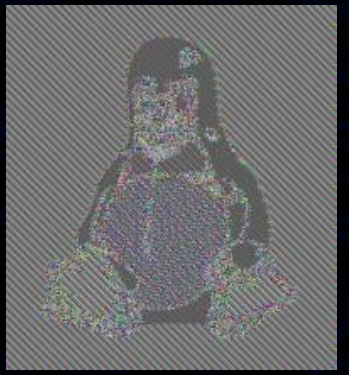
Cryptography: Symmetrical

Can be very fast on a lot of data

Use the right options

Keeping the secret secret is the hardest thing in the world (memory leak, cpu leaks, user input, program input, ..)





https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation

Cryptography: A-symmetrical

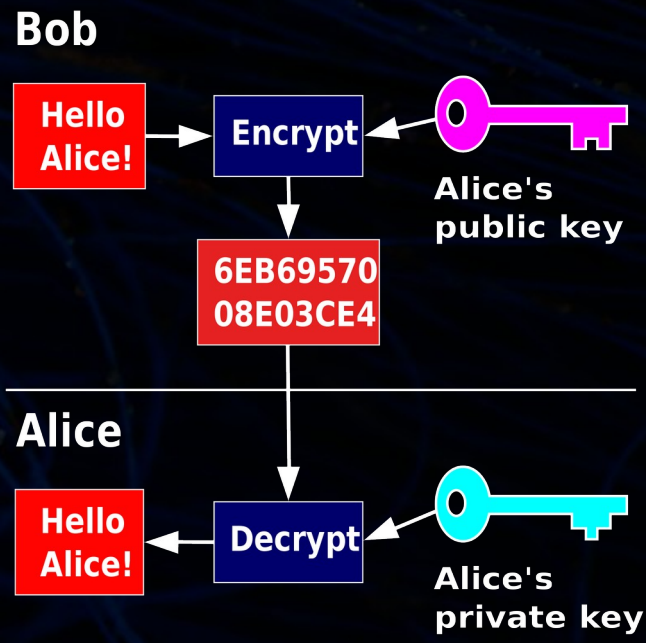
Pro: The best since sliced bread

Con: As slow as slicing bread

Keeping the key a secret is work

How do you know who the other side is?

How to agree on a shared secret?



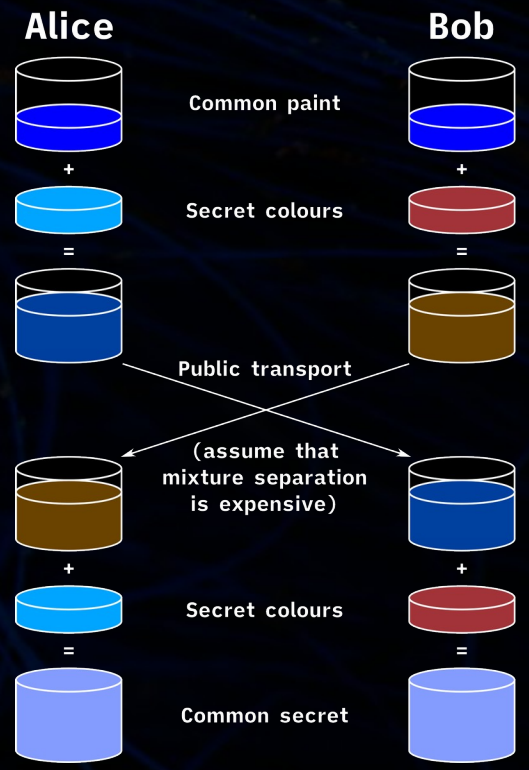
https://en.wikipedia.org/wiki/Public-key_cryptography

Cryptography: Diffie Hellman

Clever way to get common secret

Paint illustrates it

Software does it.



https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange

Public/Private: RSA v.s. EC

RSA numbers look great ^W large

EC numbers look small (256 bits)

RSA uses prime numbers

Not all numbers are primes.

Finding primes is very hard

Getting just random bits (EC) is very fast

Security Strength	RSA key length
<= 80	1024
112	2048
128	3072
192	7680
256	15360

https://www.jscape.com/blog/should-i-start-using-4096-bit-rsa-keys

What’s that EC?

Elyptic Curves – They are new (since 2006)

Used everywhere on the internet (hello IPv6?)

Smaller

Faster

Different Curves

Symmetric	ECC	DH/DSA/RSA
80	163	1024
112	233	2048
128	283	3072
192	409	7680
256	571	15360

Table 1: Comparable Key Sizes (in bits)

RFC 4492

The curves

Most are from NIST (famous NSA help)

25519 is the best these days

Ed25519 / X25519

https://safecurves.cr.yp.to/

		Parameters:			ECDLP security:				ECC security:			
Curve	Safe?	field	equation	base	rho	transfer	disc	rigid	ladder	twist	complete	ind
Anomalous	False	True ✔	True ✔	True ✔	True ✔	False	False	True ✔	False	False	False	False
M-221	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔
E-222	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔
NIST P-224	False	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	False	False	False	False	False
Curve1174	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔
Curve25519	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔
BN(2,254)	False	True ✔	True ✔	True ✔	True ✔	False	False	True ✔	False	False	False	False
brainpoolP256t1	False	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	False	False	False	False
ANSSI FRP256v1	False	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	False	False	False	False	False
NIST P-256	False	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	False	False	True ✔	False	False
secp256k1	False	True ✔	True ✔	True ✔	True ✔	True ✔	False	True ✔	False	True ✔	False	False
E-382	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔
M-383	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔
Curve383187	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔
brainpoolP384t1	False	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	False	True ✔	False	False
NIST P-384	False	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	False	False	True ✔	False	False
Curve41417	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔
Ed448-Goldilocks	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔
M-511	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔
E-521	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔	True ✔

What operations?

Encrypting

Decryption

Sign

Verify

Operations: the real basics

A public key is always a calculation from the private key

ALWAYS. That’s why they are a pair

The other way around doesn’t work.

It’s safe to publish the public key

Encrypt / Decrypt (Sym)

- Is just calculating with large numbers:
  - the secret key
- Message limited to size of the number
- Repeat if necessary (blocks)

Encrypt (A-Sym)

- Is just calculating with large numbers:
  - the private key
- Message limited to size of the number
- Repeat if neccesary (blocks)

Decrypt (A-Sym)

Is just calculating with large numbers:
    the public key

Message limited to size of the number

Repeat if necessary (blocks)

Sign

- Is just calculating with large numbers:
  - the private key
- Message limited to size of the number
- Calculate a hash from the document
- Encrypt the hash for the public key

Verify

Is just calculating with large numbers:
    the public key

Message limited to size of the number

Calculate a hash from the document

Decrypt the hash as signed with the public key

Compare the 2 hashes

Show me: The private key

```
Generating RSA private key, 512 bit long modulus (2 primes)
...++++++++++++++++++++++++++
.++++++++++++++++++++++++++
e is 65537 (0x010001)
-----BEGIN RSA PRIVATE KEY-----
MIIBOgIBAAJBAL4gBtVIgqm59JqxKYRy3UOX4QUeOyjkO3KwaPyH841yEuQkK+dZ
gSskYwk9T2rYkEJTk1zLHYtQEgACjkxiwC0CAwEAAQJAaR80KAFrnYSf5lhGaGFK
TfFQDZBljXqEFCnOIrLYyz9uuCuls5bkjShp6Bzj6ZvpsZtQsR2MeX2/PxnO6HVP
gQIhAO/huMbkeVcuNlQmlyRvzrK9lRS1n/KHvCMbnto9mIwlAiEAyuZtGuvIn4SC
vIGPG7P3yMs3JS5t9iXXL1EuDZ+WoWkCIE73WWJv99nRJqVRBtRB0iNY8Wid2Yd3
U2GjSAnD2SHZAiAEZXG/v5QdQLXEd9ot83xO8XhEafkf+DntYIjq6qZM4QIhALRx
rvwT1bRBynyrvH+lphMOuNpRNSPauneAYQnBm8qZ
-----END RSA PRIVATE KEY-----
```

```
mendel@laborans:~> openssl rsa -in rsa.prv  -noout -text
RSA Private-Key: (512 bit, 2 primes)
modulus:
    00:be:20:06:d5:48:82:a9:b9:f4:9a:b1:29:84:72:
    dd:43:97:e1:05:1e:3b:28:e4:3b:72:b0:68:fc:87:
    f3:8d:72:12:e4:24:2b:e7:59:81:2b:24:63:09:3d:
    4f:6a:d8:90:42:53:93:5c:cb:1d:8b:50:12:00:02:
    8e:4c:62:c0:2d
publicExponent: 65537 (0x10001)
privateExponent:
    69:1f:34:28:01:6b:9d:84:9f:e6:58:46:68:61:4a:
    4d:f1:50:0d:90:65:8d:7a:84:14:29:ce:22:b2:d8:
    cb:3f:6e:b8:2b:a5:b3:96:e4:8d:28:69:e8:1c:e3:
    e9:9b:e9:b1:9b:50:b1:1d:8c:79:7d:bf:3f:19:ce:
    e8:75:4f:81
prime1:
    00:ef:e1:b8:c6:e4:79:57:2e:36:54:26:97:24:6f:
    ce:b2:bd:95:14:b5:9f:f2:87:bc:23:1b:9e:da:3d:
    98:8c:25
prime2:
    00:ca:e6:6d:1a:eb:c8:9f:84:82:bc:81:8f:1b:b3:
    f7:c8:cb:37:25:2e:6d:f6:25:d7:2f:51:2e:0d:9f:
    96:a1:69
exponent1:
    4e:f7:59:62:6f:f7:d9:d1:26:a5:51:06:d4:41:d2:
    23:58:f1:68:9d:d9:87:77:53:61:a3:48:09:c3:d9:
    21:d9
exponent2:
    04:65:71:bf:bf:94:1d:40:b5:c4:77:da:2d:f3:7c:
    4e:f1:78:44:69:f9:1f:f8:39:ed:60:88:ea:ea:a6:
    4c:e1
coefficient:
    00:b4:71:ae:fc:13:d5:b4:41:ca:7c:ab:bc:7f:a5:
    a6:13:0e:b8:da:51:35:23:da:ba:77:80:61:09:c1:
    9b:ca:99
```

Show me: The public key

```
mendel@laborans:~> openssl rsa -in rsa.prv  -pubout
writing RSA key
-----BEGIN PUBLIC KEY-----
MFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAL4gBtVIgqm59JqxKYRy3UOX4QUeOyjk
O3KwaPyH841yEuQkK+dZgSskYwk9T2rYkEJTk1zLHYtQEgACjkxiwC0CAwEAAQ==
-----END PUBLIC KEY-----
```

ASN.1 JavaScript decoder

```
SEQUENCE  (2 elem)
  SEQUENCE  (2 elem)
    OBJECT IDENTIFIER  1.2.840.113549.1.1.1  rsaEncryption  (PKCS #1)
    NULL
  BIT STRING  (592 bit) 001100000100100000000010010000010000000010111110001000000000110110101…
    SEQUENCE  (2 elem)
      INTEGER  (512 bit) 995765969004249386547973107399269947784693597609350529249961693523683O…
      INTEGER  65537
```

https://lapo.it/asn1js/

Show me: Sign some data

```
mendel@laborans:~> echo -n 'Hello World' |
> openssl dgst -sha256 -sign  rsa.prv  -hex
(stdin)= b88e2c1d17023f211c04912ecb8dc361e88ecfd162fcccba2cff954defa6cfe8
28cd3d85d99b03465a4c71dd35e7c38f492bf12d3f30aa2f47dd2aff5d7ffd5d
mendel@laborans:~>
```

```
mendel@laborans:~> echo -n 'Hello World' | sha256sum
a591a6d40bf420404a011733cfb7b190d62c65bf0bcda32b57b277d9ad9f146e  -
```

Show me: The decryption

```
>>> from Crypto.Util import number
>>> from Crypto.PublicKey import RSA
>>> pubkey = "-----BEGIN PUBLIC KEY-----\nMFwwDQYJKoZIhvcNAQEBBQADSwAwSAJ
BAL4gBtVIgqm59JqxKYRy3UOX4QUeOyjk\nO3KwaPyH841yEuQkK+dZgSskYwk9T2rYkEJTk1
zLHYtQEgACjkxiwC0CAwEAAQ==\n-----END PUBLIC KEY-----\n"
>>> rsapub = RSA.importKey(pubkey)
>>> rsapub.n
995765969004249386547973107399269947784693597609350529249961693523683084237
37378043804246792713807113471504159881027009610406541602577584514778955769516
8557
>>> rsapub.e
65537
>>> signature = 0xb88e2c1d17023f211c04912ecb8dc361e88ecfd162fcccba2cff954defa6cfe
828cd3d85d99b03465a4c71dd35e7c38f492bf12d3f30aa2f47dd2aff5d7ffd5d
>>> pow(signature,rsapub.e,rsapub.n)
409173825987017733751648542997566029938148046617392981389751408119740010069825
5623945531061987614896158256796962829914166353541427091401271197973269958
>>> number.long_to_bytes(pow(signature,rsapub.e,rsapub.n)).hex()
'01ffffffffffffffffffff003031300d060960864801650304020105000420a591a6d40bf420404a011733cfb7b190d62c65bf0bcda32b57b277d9ad9f146e'
```

Show me what happened

```
>>> number.long_to_bytes(pow(signature,rsapub.e,rsapub.n)).hex()
'01ffffffffffffffffffff003031300d060960864801650304020105000420a591a6d40b
f420404a011733cfb7b190d62c65bf0bcda32b57b277d9ad9f146e'
>>> hash = 0xa591a6d40bf420404a011733cfb7b190d62c65bf0bcda32b57b277d9ad9f
146e
>>> len('a591a6d40bf420404a011733cfb7b190d62c65bf0bcda32b57b277d9ad9f146e
')
64
>>> len('01ffffffffffffffffffff003031300d060960864801650304020105000420a5
91a6d40bf420404a011733cfb7b190d62c65bf0bcda32b57b277d9ad9f146e')
126
>>> 512 / 4
128.0
>>>
```

It’s all ASN1 inside ASN1

ASN1 is used everywhere, so the padded thing is too.

ASN.1 JavaScript decoder

```
SEQUENCE  (2 elem)
  SEQUENCE  (2 elem)
    OBJECT IDENTIFIER  2.16.840.1.101.3.4.2.1  sha-256  (NIST Algorithm)
    NULL
  OCTET STRING  (32 byte) A591A6D40BF420404A011733CFB7B190D62C65BF0BCDA32B57B277D9AD9F146E
```

```
mendel@laborans:~> echo -n 'Hello World' | sha256sum
a591a6d40bf420404a011733cfb7b190d62c65bf0bcda32b57b277d9ad9f146e  -
```

But that’s POW

Yes, large number “powers of” are slow

Computers are efficient

For verify only a hash is needed, they are not that large

Hashing can be very fast depending on the hash

And very insecure depending on the hash

Encrypting: Nested usage

- Encrypt the document with symmetrical encryption
- Use a random password for this
- Encrypt this random password for the others public key
- Include the encrypted password into the document

How to do that?

- Use libraries
- Libsodium
  - No wrong defaults
- Openssl:
  - CMS
  - SMIME

Libsodium?

It is a portable, cross-compilable, installable, packageable fork of NaCl, with a compatible API, and an extended API to improve usability even further.

https://doc.libsodium.org/

- .NET: NSec
- .NET: libsodium-core (maintenance mode)
- .NET: ASodium
- Ada: libsodium-ada
- Ada: sodiumada
- Clojure: caesium
- Crystal: Sodium
- D: LibsodiumD
- Dart (Flutter): Flutter-Sodium
- Erlang: ENaCl
- Erlang: Erlang-libsodium
- Go: Sodium
- Hack: Nuxed Crypto
- Haskell: Saltine
- Haskell: hs-sodium
- Haskell: haskell-crypto
- Haskell: cryptography-libsodium
- Java (Java Native Access): libsodium-jna
- Java (Android): Lazysodium for Android
- Java: Apache Tuweni (crypto module)
- Java: Lazysodium for Java
- JavaScript (compiled to pure JavaScript): libsodium.js
- JavaScript (compiled to pure JavaScript): js-nacl
- JavaScript (libsodium.js wrapper for browsers): Natrium Browser
- JavaScript (NodeJS): sodium-native
- JavaScript (NodeJS): sodium
- Kotlin Multiplatform: kotlin-multiplatform-libsodium
- Lua: luasodium
- PHP: libsodium-php
- PHP: dhole-cryptography
- Pharo 7/8: Crypto-Nacl
- Pony: Pony-Sodium
- Python: LibNaCl
- Python: PyNaCl
- Python: PySodium
- R: Cyphr
- R: Sodium
- REALbasic and Xojo: RB-libsodium
- Ruby: RbNaCl

Keep your private key private

Hardware can support you

TPM

Smartcards

Yubikey / skey/ nitrokey / …

Hardware best practice

Let the hardware calculate the keys

Do the decryption from the shared key on CPU

RSA is slow, use EC

Hardware best practice

Let the hardware calculate the keys

Do the decryption from the shared key on CPU

RSA is slow, use EC

RSA is SLOW!!!!!!! use EC!

OpenSSL has support via pkcs11

Mostly X25519 is not supported :-(

A private key is private

Inside an hardware module this counts really

If the module breaks your key is gone

Backups not always possible

Consult an expert for real world applications

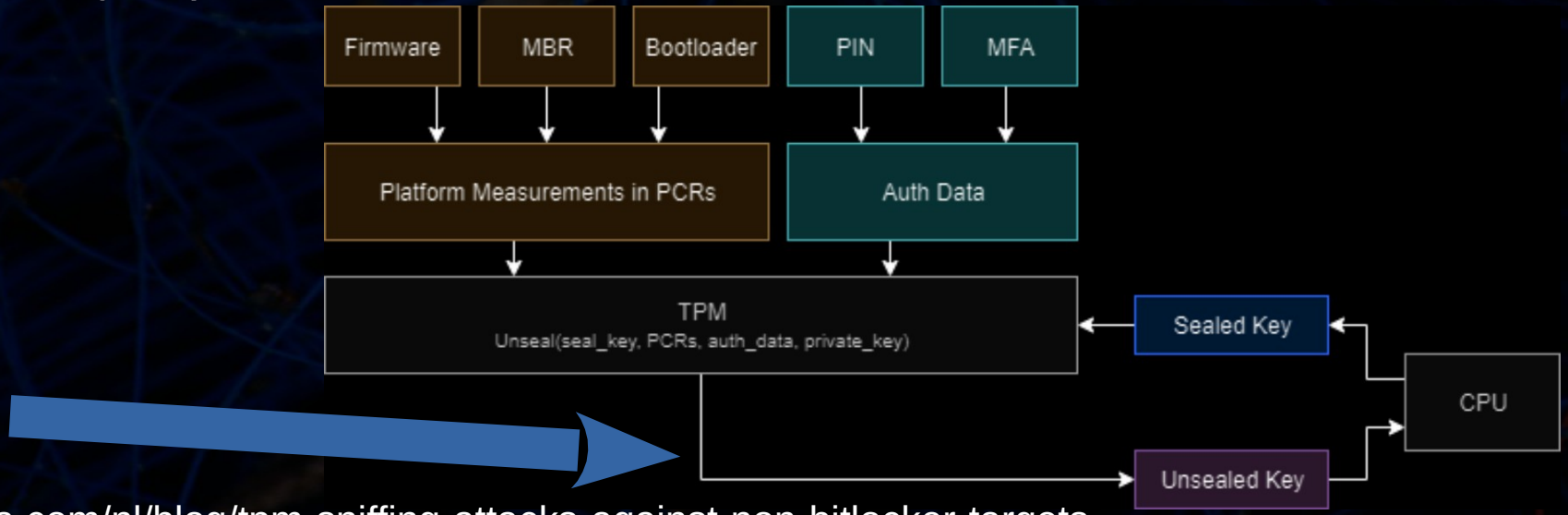
Example TPM

Sealed key means: Encrypted piece of data used as password somewhere

Real encryption key is very safe in TPM

Software requests a shared secret

Encryption key is plaintext communicated



https://www.secura.com/nl/blog/tpm-sniffing-attacks-against-non-bitlocker-targets

TLS

ECDHE_RSA_WITH_AES_256_GCM_SHA384

ECDHE

RSA

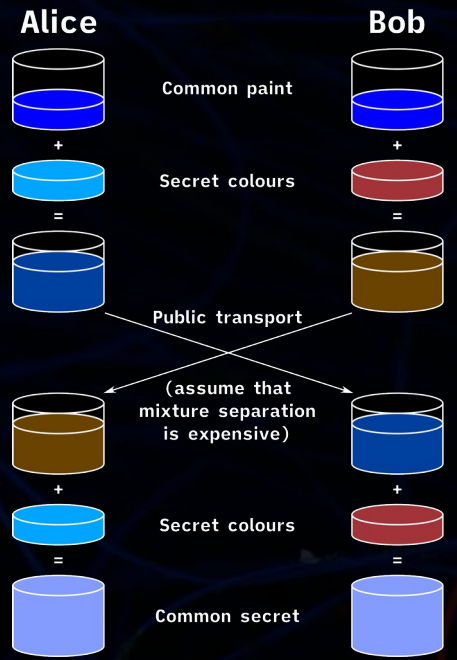
AES (256 bits) (GCM modus)

SHA384

TLS : ECDHE

ECDHE

- Both sides invent a new (ephemeral, short term) EC key
- Do Diffie-Hellman on these EC keys
- Have a shared secret in the end



TLS : RSA

Use RSA for authentication (certificates)

TLS: AES256-GCM

Encryptes all the data sent over with AES 256 in GCM mode

Galois/Counter Mode

TLS: SHA384

- Uses SHA384 for hashing
  - Used for data integrety, so you know the data is not changed by errors.

What to use

Keys: EC (X25519 if possible else NIST P-256)

POLY1305 else SHA2-512

CHACHA20 or AES (GCM or CBC)

Run away if used:

RSA < 2048

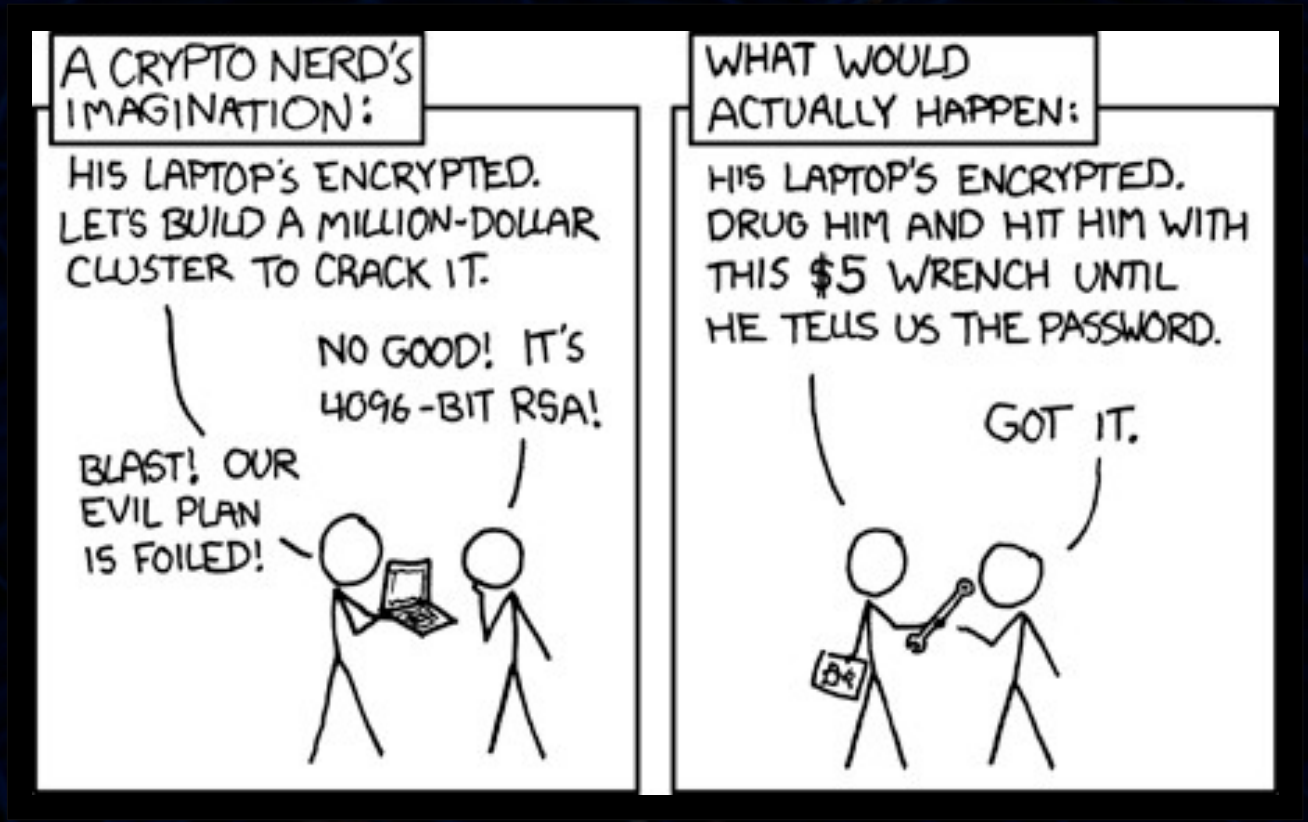
SHA1, MD5 or older

DES, 3DES, ….

No encryption

“Militairy grade” encryption

There’s a XKCD for everything



https://xkcd.com/538/

Not an XKCD for everything yet

- Revoking signatures (a public encryption made with a private key) can **not** be revoked
- The public key is public – widely known!
- The signature is public – widely known!

https://blog.cryptographyengineering.com/2020/11/16/ok-google-please-publish-your-dkim-secret-keys/

Questions

Example question:

- Is base64 encryption?