CAA 24-25

Randomness Generation

Alexandre Duc

Random Numbers

Question

Where do we need random numbers ?

What are the risks if they are not perfectly random ?

Bad Randomness is Destructive : IVs

- IVs are meant to be fresh but **reusing** them has different outcomes based on their usage.
- In CBC, detection of similarities in the first bloc.
- In CTR, CCM, GCM, and stream ciphers reusing an IV is catastrophic $\rightarrow$ XOR ciphertexts $=$ XOR plaintexts.

Bad Randomness is Destructive : RSA

- In RSA, p and q are prime numbers that are used to obtain the public modulus $n = pq$.
- Given p and q, one can recover the **private key**.
- If one prime number is reused between two moduli, one can **factor** them.

Bad Randomness is Destructive : DSA

- DSA is the digital signing algorithm which is based on the discrete logarithm problem.
- It is **non-deterministic** and uses randomness to sign.
- Reusing randomness in a signature implies recovering the **private key** (see further lecture).
- Allowed to recover the ECDSA private key in Playstation 3 used to sign software.

Bad Randomness is Destructive : Statistics

- *Mining your Ps and Qs : Detection of Widespread Weak Keys in Network Devices*, USENIX 2012.
- 0.75% of TLS certificates share keys
- Possible to recover **RSA private keys** of 0.5% of TLS hosts and 0.03% of SSH hosts.
- Possible to recover **DSA private keys** of 1.03% of SSH hosts (using only two signatures per host).
- Some devices for RSA were taking 2 random primes out of a hardcoded list of 9.

Random Number Generator

Random Number Generator

A **random number generator (RNG)** is a physical or a computational device that generates a sequence of numbers that appear to be random.

Statistical Tests

- NIST/Diehard/Dieharder statistical suite tests for "good" statistical properties of generated numbers.
- Check variance, χ^2, ... properties.

Question

Is it sufficient ?

Good PRNG ?

Question

What are the statistical properties of the following PRNG :

$$\text{output} = \text{AES}_0(\text{previous\_output})$$

Is it a good PRNG ?

Statistical Tests

- In cryptography : good statistical properties are **not enough !**
- We requires an additional property : **unpredictability** (also called **next-bit test**).

Next-Bit Test

Next-Bit Test

A random number generator is said to pass the **next-bit test** if an adversary knowing the first i output bits is unable to predict the $(i+1)$-th bit with a probability that is significantly different from $\frac{1}{2}$ within a feasible computational effort.

Theorem (Yao, 1982)

A random number generator passes the next-bit test if and only if it passes all efficient statistical tests.

- **Informal definitions**.
- The next-bit test is a theoretical test.

True Random Number Generators

True Random Number Generator

A **true random number generator (TRNG)** is an apparatus that generators random numbers from a physical process.

- Nuclear decay
- Coin tosses
- Atmospheric noise
- Quantum effects
- Thermal noise in a processor
- ...

Bad "TRNGs"

- Time / date
- pid
- Processor temperature

[Benadjila and Ebalard, Randomness of Random in Cisco ASA]

"In a system which expects to perform cryptographic tasks (...), no developer should start playing with time primitives to extract bits of entropy, expecting an happy end to the story. The system should be designed to include serious entropy sources (...)".

Source: https://eprint.iacr.org/2023/912.pdf

Entropy

- The **bit entropy** measures how much uncertainty there is in the bit.
- Linked to the **probability distribution of a bit**.
- Entropy of uniform bit distribution : 1 (max)
- Entropy of constant bit distribution : 0 (useless)

Bit Distribution Properties

Bias of a bit

A bit is **biased** if its distribution differs from the uniform distribution. For instance, if $\Pr[b = 0] = 0.6$.

Independent bits

Two bits are **independent** if $\Pr[b_1, b_2] = \Pr[b_1] \Pr[b_2]$.

Question

How to transform a source of **independent**, but **biased** random bits into **unbiased** bits ?

von Neumann Randomness Extraction

- Simple solution : transform it into an unbiased bit sequence with the **von Neumann randomness extraction** procedure.
- It works as follows : consider the bit sequence by pairs of two bits. If the pair is 00 or 11, discards them. If the pair is 01, then replace it by 0, and if the pair is 10, then replace it by 1.
- Other, more efficient randomness extractor exist. Some are based on cryptographic primitives (e.g. hash functions).

Pseudorandom Number Generators

Pseudorandom Number Generator

A **pseudorandom number generator (PRNG)** is a deterministic algorithm whose aim is to generate a sequence of numbers exhibiting good statistical properties.

- The value initializing the state of a PRNG is called a **seed**.
- As a PRNG is **deterministic**, it outputs the same sequence of random numbers when seeded with the same value.

Bad PRNGs : Mersenne Twister

Wikipedia

The Mersenne Twister is a pseudorandom number generator (PRNG). It is by far the most widely used general-purpose PRNG.[. . .]
Implementations generally create random numbers faster than other methods.

- A very long period of $2^{19937} - 1$.
- The Mersenne Twister is **not secure** and should **never** be used in cryptographic applications.
- Given 624 outputs, one can recover the inner state.

Bad PRNGs : Common Libraries

Libraries

Most of the PRNGs provided by default by common libraries (Boost, GMP, ...) and languages (C/C++, PHP, Python, etc.) are **not cryptographically secure**.

Examples of **bad calls** :

- **C/C++** : `rand()`, `rand48()`, `arc4random()`
- **Java** : `java.Util.Random` / `Math.random()`
- **Python** : random

Warning

Read the documentation to see if it is cryptographically secure !

Bad PRNGs : Others

Never use the following as PRNGs :

- RC4 (stream cipher but broken)
- Lagged Fibonacci, LFSR, Lehmer, ... (mostly linear or affine)
- `random.org`, `randomnumbers.info`, ...

Cryptographically Secure PRNGs

Cryptographically Secure PRNGs

A **cryptographically secure pseudorandom number generator (CPRNG)** is a deterministic algorithm whose aim is to generate a sequence of numbers that is unpredictable for an adversary within a feasible amount of computations.

- Many implementations of CPRNGs are available.
- Stream ciphers are supposed to be CPRNGs.
- Some are based on hash functions, block ciphers, or number-theoretic hard problems.

Blum Blum Shub PRNG

- The Blum-Blum Shub PRNG is cryptographically secure if the problem of factoring large composite numbers is hard.
- It works as follows :
  - **Initialization** : Generate two sufficiently large prime numbers p and q that are congruent to 3 (mod 4), compute $n = p \cdot q$ and throw away p and q.
  - **Random Bit Sequence Generation** : Given a seed x_0 uniformly distributed on $[2, n-1]$, compute the sequence $x_i = x_{i-1}^2 \bmod n$ and output the bit $\mathrm{lsb}(x_i)$.
- The Blum Blum Shub PRNG is **extremely slow** in practice.

Blum-Micali PRNG

- The Blum-Micali PRNG is cryptographically secure if the discrete logarithm problem is hard.
- It works as follows :
  - **Initialization** : Generate a sufficiently large prime number p and find g a generator of $\mathbb{Z}_p^*$.
  - **Random Bit Sequence Generation** : Given a seed x_0 uniformly distributed on $[2, p-1]$, compute the sequence $x_{i+1} = g^{x_i} \bmod p$ and output 1 if $x_i \leq \frac{p-1}{2}$ and 0 otherwise.
- The Blum-Micali PRNG is **extremely slow** in practice.

Forward and Backward Security

Forward Security (PRNG)

The leakage of the inner state of the PRNG should not compromise the **previously** outputted random bits.

Backward Security (PRNG)

It should be possible for future outputs to be secure if current state is compromised.

Backward Security

Backward Security requires injecting fresh entropy.

- **Seeding** operation
- **Reseeding** operation
- **Value generation**

Sources of Entropy

- To seed a PRNG, we need **entropy sources**
- It is also needed to perform reseeding operations.

Question

On a computer, what would you use as entropy source ?

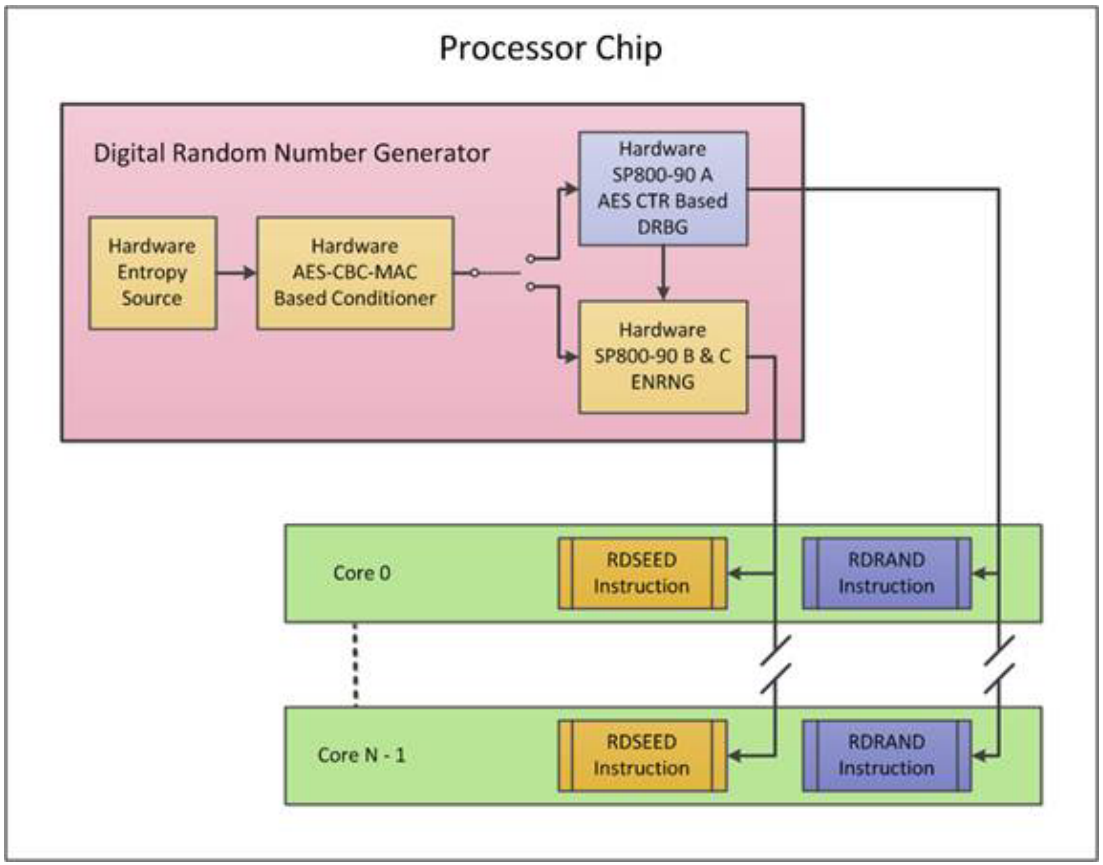
/dev/random and /dev/urandom

- On Unix-like operating systems, `/dev/random` is a special file that acts as a cryptographically secure PRNG. It collects environmental noise from device drivers (events related to disks, network, mouse, etc.)
- Since kernel 5.6, `/dev/random` is only blocking until the CPRNG is properly initialized.
- `/dev/urandom` is a **non-blocking** variant of `/dev/random`. It means that it outputs pseudorandom bytes, even if insufficient randomness is available. In extreme cases, it can become predictable.
- The presence/absence of both `/dev/random` and `/dev/urandom` and their implementation is varying among the different flavours and versions of operating systems.

Intel's RDSEED and RDRAND

- Embedded CPRNG on Intel CPUs.
- AMD decided to implement the same instructions.
- Two instructions are available :
  - RDRAND : get high quality random data. PRNG regularly reseeded. Faster.
  - RDSEED : get a high quality random seed for a software CPRNG. TRNG. Slower.

Intel's RDSEED and RDRAND



Source : https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide

RNRAND/RDSEED is not perfect

Documentation of RDSEED

The **Carry Flag** indicates whether a random value **is available** at the time the instruction is executed. CF=1 indicates that the data in the destination is valid. Otherwise CF=0 and the data in the destination operand will be returned as zeros for the specified width.

- 2020 : CrossTalk vulnerability. Allows to read values of RDRAND, RDSEED across cores.
- Side-channel attack.
- Mitigations -> 3% of original speed

Pool of Entropy

- An operating system typically keeps an entropy pool which is regularly fed with fresh entropy.

Question

Linux XORs the fresh entropy into the pool. Is it a good idea ?
What if an attacker can access the RAM ?

Mixing Different Sources

Solution

Using a simple XOR works but is not the best idea. First, it is important to notice that if the attacker cannot read the value of the pool, entropy cannot decrease in the pool. However, someone generating random values and having access to the RAM can control its random value to control the whole pool. It is better to use a hash function, for instance, using $H(\text{old}\|\text{new})$ as a new pool (which is used for instance in OpenSSL).

NIST SP800-90A

- NIST standardized 4 (in fact 3) deterministic random byte generators (DRBG).
- We present here only the **general picture**.

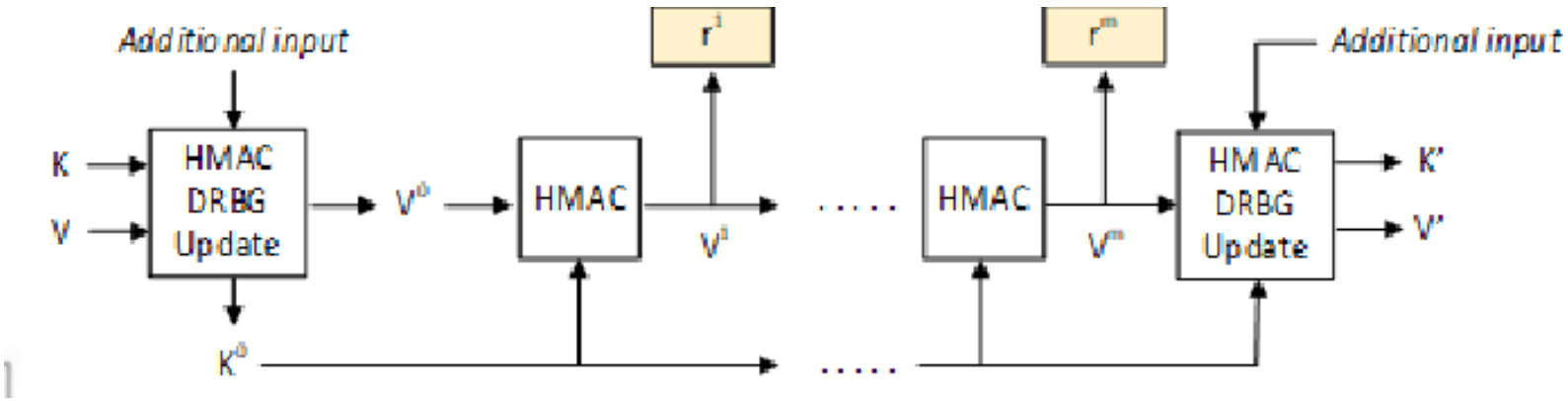
Warning

Read the standard if you want to implement it.

Hash_DRBG

- Based on the use of a hash function.
- Idea : $w = H(\text{state})$. state $=$ state $+ w$.
- Supports SHA-1 and SHA-2.
- Standardized in NIST SP800-90A (§10.1.1)

HMAC_DRBG



Source : http://ijns.jalaxy.com.tw/contents/ijns-v23-n1/ijns-2021-v23-n1-p33-41.pdf

HMAC_DRBG

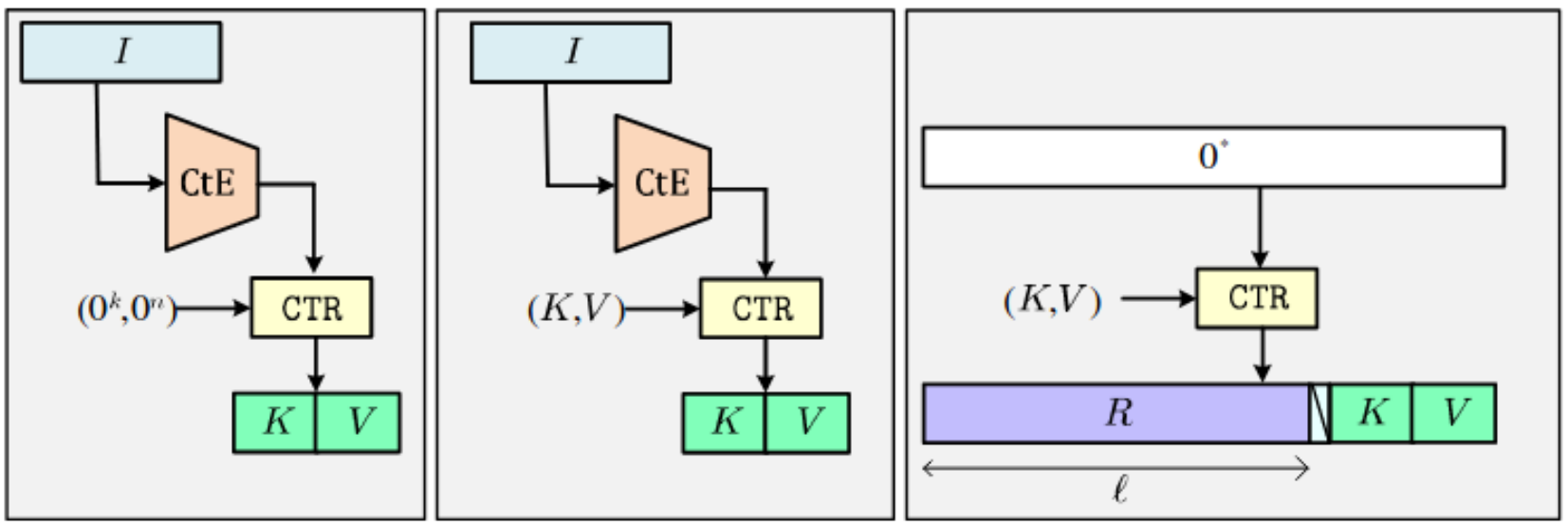
- Based on HMAC :
- Idea : Obtain key K and state V from randomness. To generate, $V \leftarrow \text{HMAC}(K, V)$. Update V and K using HMAC and additional randomness.
- Standardized in NIST SP800-90A (§10.1.2)
- Slower than Hash_DRBG but more secure.

CTR_DRBG

- Based on the use of a block cipher operated in counter mode
- Supports either 3-key Triple-DES or AES-128, AES-192 and AES-256.
- Idea : Obtain key K and state V from randomness. To generate, use CTR with key K on state V to obtain random values. Update V and K using CTR.
- Standardized in NIST SP800-90A (§10.1.3)

CTR_DRBG

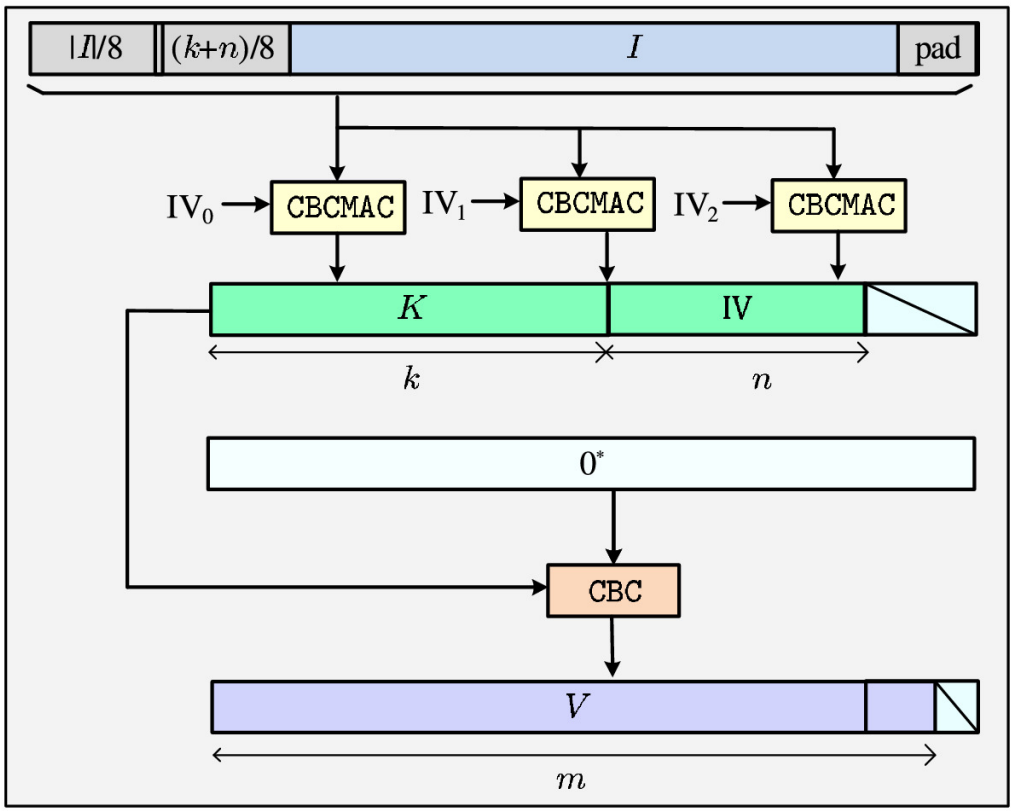
procedure setup$^E(I)$	**procedure** refresh$^E(S, I)$	**procedure** next$^E(S, \ell)$
$X \leftarrow \mathrm{CtE}[E](I)$	$X \leftarrow \mathrm{CtE}[E](I)$	$K \leftarrow S[1:k];\ V \leftarrow S[k+1:k+n]$
$K \leftarrow 0^k;\ \mathrm{IV} \leftarrow 0^n$	$K \leftarrow S[1:k]$	$r \leftarrow n \cdot \lceil \ell/n \rceil$
$S \leftarrow \mathsf{CTR}_K^{\mathrm{IV}}[E](X)$	$V \leftarrow S[k+1:k+n]$	$P \leftarrow \mathsf{CTR}_K^V[E](0^{r+k+n})$
return S	$S \leftarrow \mathsf{CTR}_K^V[E](X)$	$R \leftarrow P[1:\ell]$
	return S	$S \leftarrow P[r+1:r+k+n]$
		return (R, S)



Source : Hoang and Shen, Crypto 2020

CTR_DRBG : CtE

procedure $\mathrm{CtE}[E, m](I)$
$X \leftarrow \mathrm{pad}\big([|I|/8]_{32} \,\|\, [(k+n)/8]_{32} \,\|\, I\big)$
for $i \leftarrow 0$ **to** 2 **do**
  $\mathrm{IV}_i \leftarrow \pi([i]_{32} \,\|\, 0^{n-32})$; $T_i \leftarrow \mathsf{CBCMAC}^{\mathrm{IV}}[\pi](X)$
$Y \leftarrow T_1 \,\|\, T_2 \,\|\, T_3$; $K \leftarrow Y[1:k]$; $\mathrm{IV} \leftarrow Y[k+1:k+n]$
$C \leftarrow \mathsf{CBC}^{\mathrm{IV}}_K[E](0^{3n})$; **return** $C[1:m]$



Source : Hoang and Shen, Crypto 2020

DUAL_EC_DRBG

- Published by NIST in 2006 in NIST SP-800-90A in collaboration with NSA. Removed in 2012.
- Only proposal in NIST SP-800-90A based on mathematics $\rightarrow$ thousand time slower.
- Uses elliptic curves with parameters P, Q coming out of nowhere.
- Too many bits are output : 0.1% bias guessing the next bit.
- Potential **backdoor** : knowing e such that $Q = eP$ would allow to predict all the future outputs.

Randomness Generation in Practice

- **C/Linux :system** : `getrandom()` system call (`getentropy()` on BSD). Reads from `urandom` if has a high enough entropy level. Non blocking afterwards.
- **C++/Windows** : `BCryptGenRandom`. Possibility to choose algorithm via **providers**. Default : `CTR_DRBG`.
- **C/C++ libraries** : OpenSSL : `RAND_bytes` (hard to use correctly). Libsodium : `https://libsodium.gitbook.io/doc/generating_random_data`
- **Java** : `java.security.SecureRandom`. Possibility to choose algorithm via providers. Default : `SHA1PRNG`.
- **Python** : `secrets` module. Uses underlying OS RNG.
- **Rust** : `rand::rngs::OSRng`. Takes randomness from OS. `rand::rngs::StdRng` is a CPRNG. Uses currently Chacha20.

Warning

Warning

In all these cases, **read** the documentation and **check** the return values !

Generating Random Numbers

- Typically, a CPRNG will deliver various amounts of random bytes. This is fine when generating parameters for symmetric cryptography.
- However, when dealing with public-key algorithms, one is often asked to generate uniformly distributed random numbers that are strictly smaller than an arbitrary number n.
- Going from random bytes to a random number is a delicate operation.

Generating Random Numbers

Question

How would you generate a random integer modulo 18 (i.e., a random number in $\mathbb{Z}_{18}$)?

Wrong Methods to Generate Random Numbers

We want to generate a uniform number smaller than n, where n is an ℓ-bit number.

- one generates an ℓ-bit number and reduces it modulo n ;
- one generates an ℓ-bit number r and if $r > n$, then one clears the most significant bit ;
- one generates an $\ell - 1$-bit number (i.e., computing $r \bmod 2^{\ell-1}$).

The Right Way : the Rejection Method

- The right way is to employ a **rejection method**. To generate a number $r < n$ uniformly at random, where n is an ℓ-bit number, one proceeds as follows :
  1. Generate an ℓ-bit string r uniformly at random.
  2. If $r < n$, then output it. Otherwise, jump to 1.
- The loop will be taken a variable number of times, but it will quickly finish with high probability.

Question

What is the probability of failure in the worst case for one iteration ?

The Debian Fiasco

- From 2006 to 2008 on Debian : completely broken OpenSSL PRNG.
- Only 32768 possible SSH keys.
- Combination of **bad software design** and **too clever undocumented tricks**.

OpenSSL PRNG

- Gathers entropy from different sources (e.g. `/dev/urandom`) and combines them using a hash function.
- Hash function is using Merkle-Damgård construction. Buffer is compressed into state : `MD_Update(&m,buf,j);`
- `RAND_add(buf, n, e)` function : adds a buffer of size n into the state with e bits of entropy.
- Allows to update the **entropy estimate**.

Problematic Code ?

```
char buf[100];
fd = open("/dev/random", O_RDONLY);
n = read(fd, buf, sizeof buf);
close(fd);
RAND_add(buf, sizeof buf, n);
```

Other Occurrence

```
i=fread(buf,1,n,in);
if (i <= 0) break;
/* even if n != i, use the full array */
RAND_add(buf,n,i);
```

Obtaining an Output

- Function `RAND_bytes(buf, sizeof buf)`.
- Adds the values of the buffer to the entropy pool before obtaining output.
-

```
char buf[100];
n = RAND_bytes(buf, sizeof buf);
```

- Valgrind got crazy : conditional jump based on an uninitialized value.

```
When debbuging applications that make use of openssl using
valgrind, it can show alot of warnings about doing a conditional
jump based on an unitialised value.  Those unitialised values are
generated in the random number generator.  It's adding an
unintialiased buffer to the pool.

The code in question that has the problem are the following 2
pieces of code in crypto/rand/md_rand.c:

247:
                MD_Update(&m,buf,j);

467:
#ifndef PURIFY
                MD_Update(&m,buf,j); /* purify complains */
#endif

...

What I currently see as best option is to actually comment out
those 2 lines of code.  But I have no idea what effect this
really has on the RNG.  The only effect I see is that the pool
might receive less entropy.  But on the other hand, I'm not even
sure how much entropy some unitialised data has.

What do you people think about removing those 2 lines of code?
```

Answers

OpenSSL dev :

```
> But on the other hand, I'm not even
> sure how much entropy some unitialised data has.
>
Not much. If it helps with debugging, I'm in favor of removing them.
(However the last time I checked, valgrind reported thousands of bogus
error messages. Has that situation gotten better?)
```

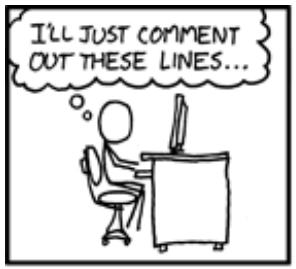
Not an OpenSSL dev :

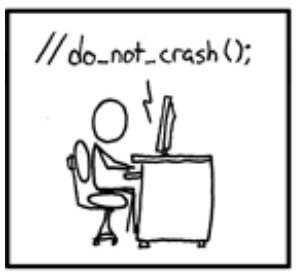
```
I recently compiled vanilla OpenSSL 0.9.8a with -DPURIFY=1 and on Debian
GNU/Linux 'sid' with valgrind version 3.1.1 was able to debug some
application using both TLS/SSL as S/MIME without any warning or error
about the OpenSSL code. Without -DPURIFY you're indeed flooded with
warnings.

So yes I think not using the uninitialized memory (it's only a single
line, the other occurrence is already commented out) helps valgrind
```

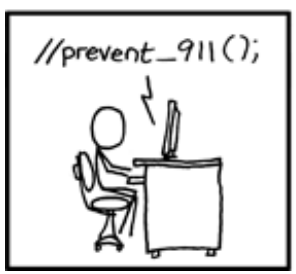
Result

- The only randomness introduced in the PRNG is the PID of the process.
- The problem comes from partially documented clever code.
- We also had duplicated code with different comments
- More infos on `https://research.swtch.com/openssl`

Conclusion







IN THE RUSH TO CLEAN UP THE DEBIAN-OPENSSL FIASCO, A NUMBER OF OTHER MAJOR SECURITY HOLES HAVE BEEN UNCOVERED:

AFFECTED SYSTEM	SECURITY PROBLEM
FEDORA CORE	VULNERABLE TO CERTAIN DECODER RINGS
XANDROS (EEE PC)	GIVES ROOT ACCESS IF ASKED IN STERN VOICE
GENTOO	VULNERABLE TO FLATTERY
OLPC OS	VULNERABLE TO JEFF GOLDBLUM'S POWERBOOK
SLACKWARE	GIVES ROOT ACCESS IF USER SAYS ELVISH WORD FOR "FRIEND"
UBUNTU	TURNS OUT DISTRO IS ACTUALLY JUST WINDOWS VISTA WITH A FEW CUSTOM THEMES

https://www.xkcd.com/424

Symmetric Cryptography Standards

Alexandre Duc

Zoo

Symmetric cryptography is a zoo. There are dozens of primitives and not all are recommended. How do you select which one to use ?

Block Ciphers

- Dozens of designs of block ciphers have been proposed in the academic literature.
- Most of them have not been thoroughly analyzed.
- A large part of them suffers from theoretical and/or practical weaknesses.
- Some of them are “Internet”-standards.

ECRYPT-CSA Recommendation

- https://www.ecrypt.eu.org/csa/documents/D5.4-FinalAlgKeySizeProt.pdf
- European cryptographic researchers that came up with recommendations for choosing cryptography (key sizes, algorithms, ...)
- Still up to date : 2018
- Other document in French from ANSSI (2021) : https://www.ssi.gouv.fr/uploads/2021/03/anssi-guide-selection_crypto-1.0.pdf

Which Block Ciphers would you Use

Primitive
Blowfish
AES
DES
Serpent
Triple-DES
Camelia

Block Cipher Recommendations

Primitive	Legacy	Future
AES	✓	✓
Camelia	✓	✓
Serpent	✓	✓
Triple-DES	✓	x
Blowfish	✓	x
DES	x	x

source of recommendations : https://www.ecrypt.eu.org/csa/documents/D5.4-FinalAlgKeySizeProt.pdf

Triple-DES

- **Block size** of **64 bits**, **key size** of **112 bits** (two-key version) and of **168 bits** (three-keys version).
- Standardized in NIST SP800-67 Revision 1 and in ISO/IEC 18033-3 :2010.
- Effective security lower than key security (2^{112} bits for three-key version)
- **Weak points** : old design, small block size, extremely slow
- Still widely deployed in the financial industry
- **Avoid it** !

AES

- **Block size** of **128 bits**, **key size** of **128, 192** and **256 bits**.
- Standardized in NIST FIPS PUB 197 and ISO/IEC 18033-3 :2010.
- **Strong points** : good security, fast on most platforms, large block size, strong design process
- **Weak point** : the encryption and decryption algorithms are rather different

Camellia

- **Block size** of **128 bits**, **key size** of **128, 192** and **256 bits**.
- Designed by researchers from Mitsubishi and NTT (Japan)
- Standardized by CRYPTREC, NESSIE and in RFC 3713 and ISO/IEC 18033-3 :2010.
- Recommended for use with S/MIME, XML, TLS/SSL, IPSec, OpenPGP, etc.
- **Strong points** : good security, reasonably fast on most platforms, large block size, not a US-approved design
- **Weak points** : less efficient that AES

Serpent

- **Block size** of **128 bits**, **key size** of **128, 192** and **256 bits**.
- Designed by Anderson, Biham, Knudsen.
- Ranked second in the AES competition.
- **Strong points** : believed to have a bigger security margin than AES.
- **Weak points** : much slower than AES.

Block Cipher Block Size

Block Size

Recommended constructions have all **block sizes** of 128 bits. Smaller block sizes should be used only under very specific and controlled situations.

Question

Why is a small block size problematic ?

Padding

- Block ciphers and some mode of operation require the plaintext to have a size **multiple of the block size**.
- This requires some **padding**
- The same paddings can also be used for MACs.

Question

How would you pad a plaintext that is not a multiple of a blocksize ?

Padding Standards

- **Bit Padding** : 10* (defined in NIST 800-38a).
- **Byte Padding : ANSI X9.23** : arbitrary data. Last byte is size of padding.
- **Byte Padding : PKCS#7** : if k bytes are missing, add k bytes with value k.
- PKCS#5 padding is a subset of PKCS#7 for small block sizes.
- They are all secure but beware of **padding oracle attacks**.

Hash Functions

- A lot of hash function designs have been proposed.
- Several designs have been badly **broken** (including MD2, MD4, MD5, SHA1 etc.)

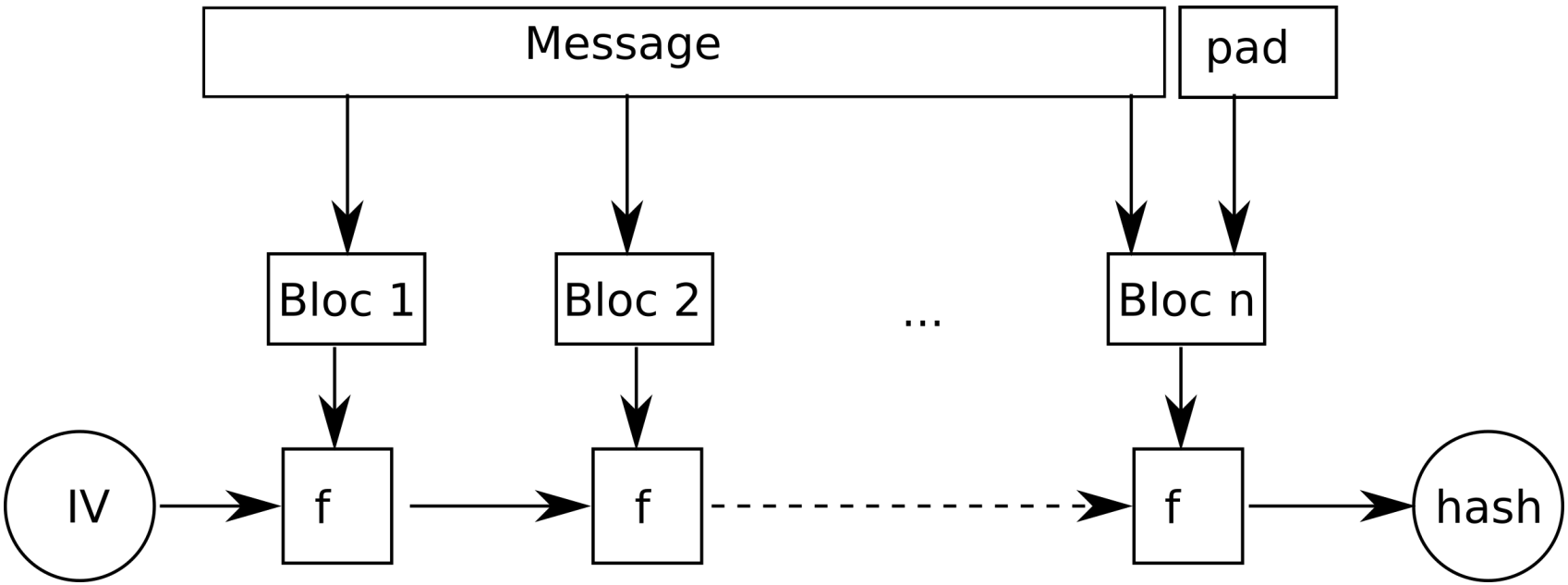
Hash Functions Recommendations

Primitive	Output length	Legacy	Future
SHA-2	256, 384, 512, 512/256	✓	✓
SHA-3	256, 384, 512	✓	✓
SHA-3	SHAKE128, SHAKE256	✓	✓
Whirlpool	512	✓	✓
Blake (1 or 2)	256, 384, 512	✓	✓
SHA-2	224, 512/224	✓	x
SHA-3	224	✓	x
RIPEMD-160	160	✓	x
MD5	128	x	x
SHA1	160	x	x

source of recommendations : https://www.ecrypt.eu.org/csa/documents/D5.4-FinalAlgKeySizeProt.pdf

Merkle-Damgård Construction

The Merkle-Damgård construction allows to transfom a **compression function** into a hash function.

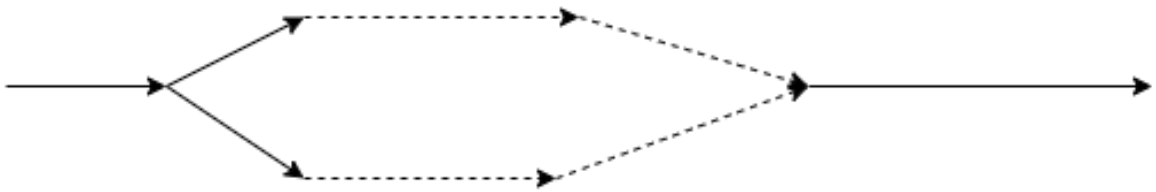


The Merkle-Damgård padding is a 1 followed by 0s follwed by the length of the message encoded over 64 bits.

SHA-1

- Standardized in NIST FIPS 180.
- Output digest of 160 bits
- SHA-1 is **broken** : one can find **collisions** within few seconds on a laptop.
- Should be avoided by all means in new applications.

Shambles Attack



- 2020 Leurent and Peyrin : Improved attack on SHA-1 : **Chosen-prefix collision attack**
- Breaks certificates, PGP, TLS, SSH
- For certificates : possible to find keys k_A and k_B such that $H(\text{Certif}(\text{Alice}, k_A)) = H(\text{Certif}(\text{Bob}, k_B))$.
- Bob asks to sign his certificate. It is a valid certificate for Alice.

SHA-2

- Standardized in NIST FIPS 180.
- Two algorithms : SHA-256 and SHA-512.
- Possible outputs digest are 224 (SHA-256 or SHA-512 truncated), 256 (normal or SHA-512 truncated), 384 (SHA-512 truncated) and 512 bits.
- Also based on the Merkle-Damgård construction.
- Unbroken for the moment.

Merkle-Damgård and Length Extension Attacks

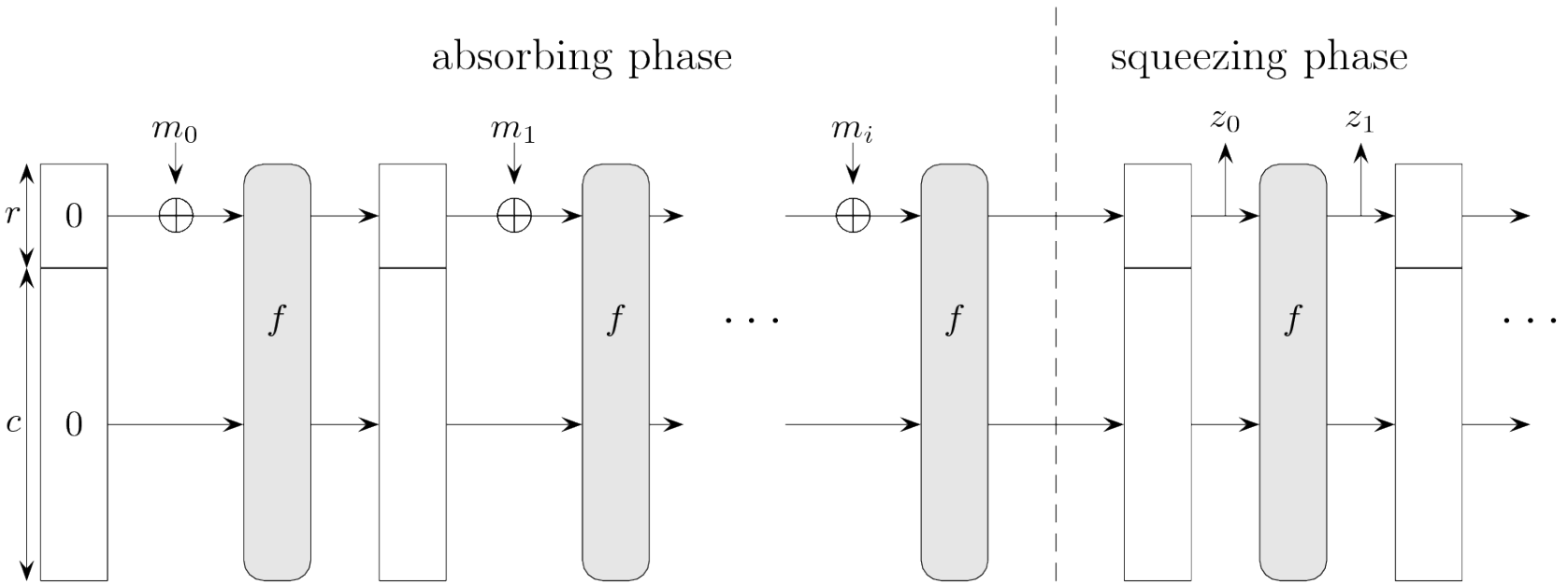
Question

Show how, in the Merkle-Damgård construction, someone can, given a hash, extend the hashed message without knowing it. In which use case is this problematic ?

SHA-3

- Result of the SHA **competition** : Keccak has been selected for becoming SHA-3
- Other finalists where : Blake, JH, Skein, Grøstl
- Standardized in NIST FIPS 202.
- Not based on Merkle-Damgård construction but on **sponge construction**.
- Versions : SHA3-224, SHA3-256, SHA3-384, SHA3-512, SHAKE128, SHAKE256
- 10^*1 padding.

Sponge Construction



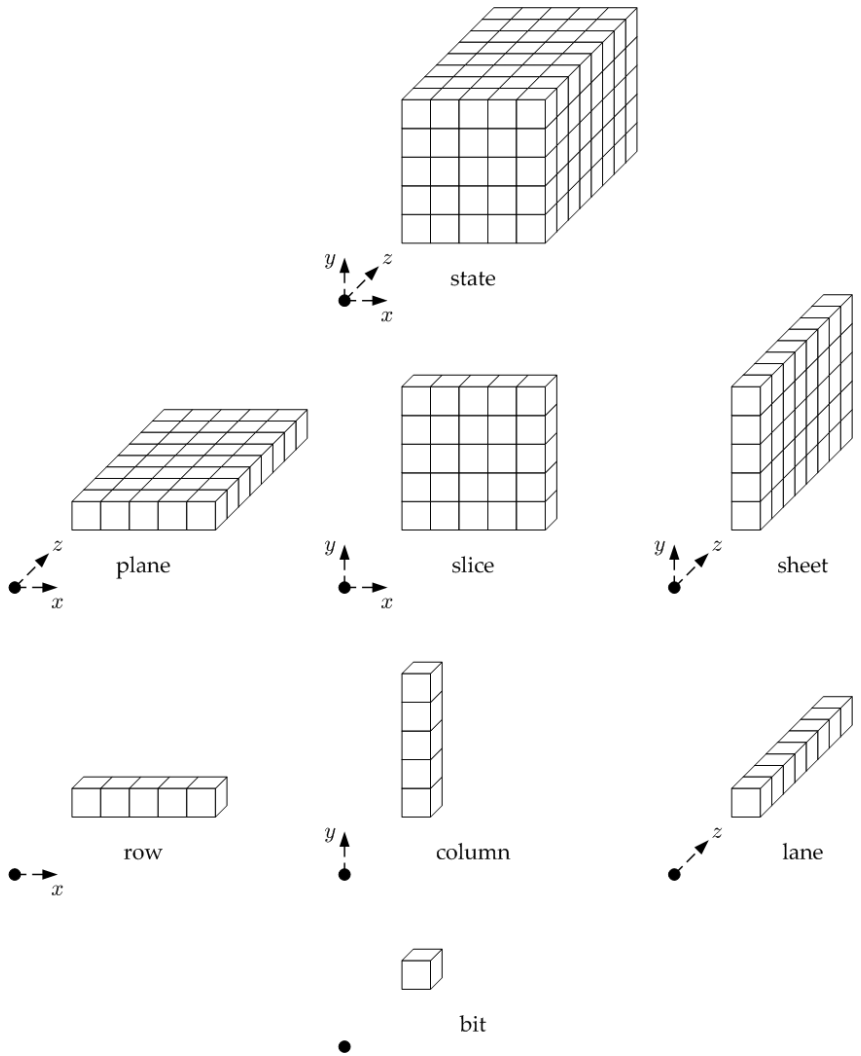
SHA3 Instances

Instance	Output	rate r	capacity c	Collision	Preimage
SHA3-224	224	1152	448	112	224
SHA3-256	256	1088	512	128	256
SHA3-384	384	832	768	192	384
SHA3-512	512	576	1024	256	512
SHAKE128	d	1344	256	$\min(d/2, 128)$	$\min(d, 128)$
SHAKE256	d	1088	512	$\min(d/2, 256)$	$\min(d, 256)$

The Keccak-f Internal Permutation

- 24 rounds R_r.
- The round operation is **easy to invert**.
- Works on a 3D state (was 2D in AES).
- Very elegant design similar to AES : each substep has a purpose.

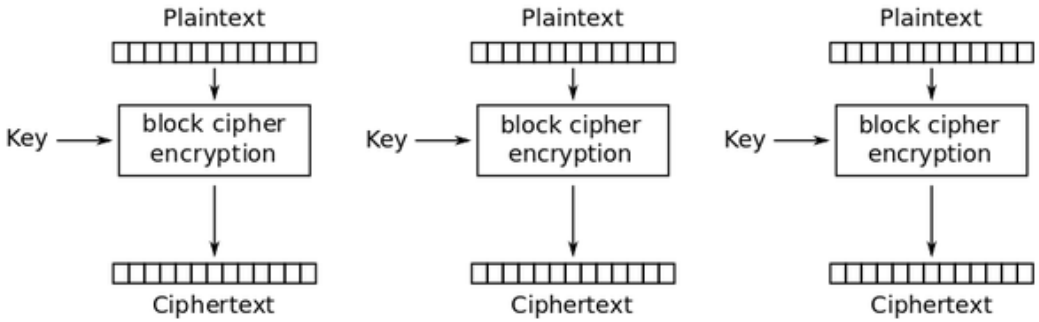
The SHA3 State



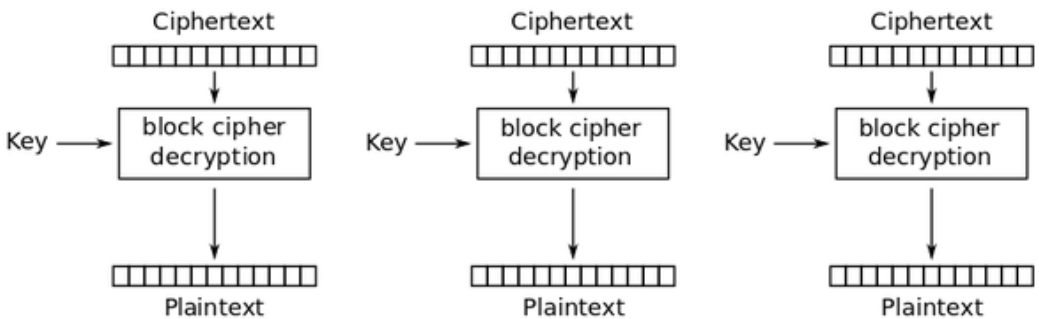
Modes of Operation

- A mode of operation allows to extend the reach of a block cipher for encrypting data of any length.
- All of them offer a security that is a function of the block length of the underlying block cipher. Due to the birthday effect, they begin to leak information after handling $2^{\frac{\ell}{2}}$ blocks, for an ℓ-bit block cipher.

ECB



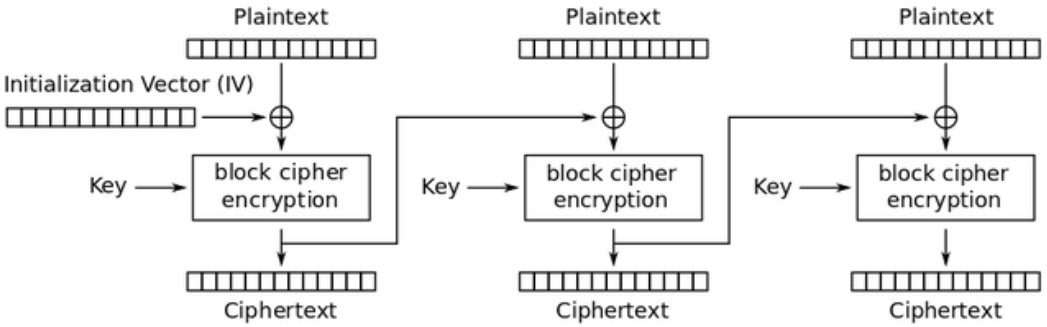
Electronic Codebook (ECB) mode encryption



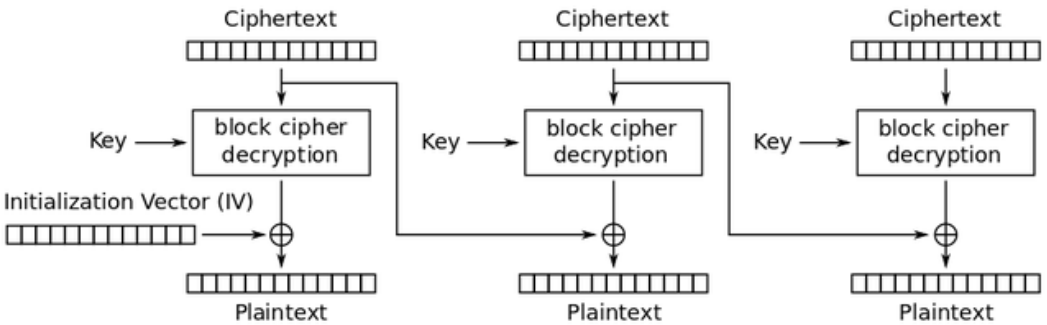
Electronic Codebook (ECB) mode decryption

Source of picture : http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation

CBC



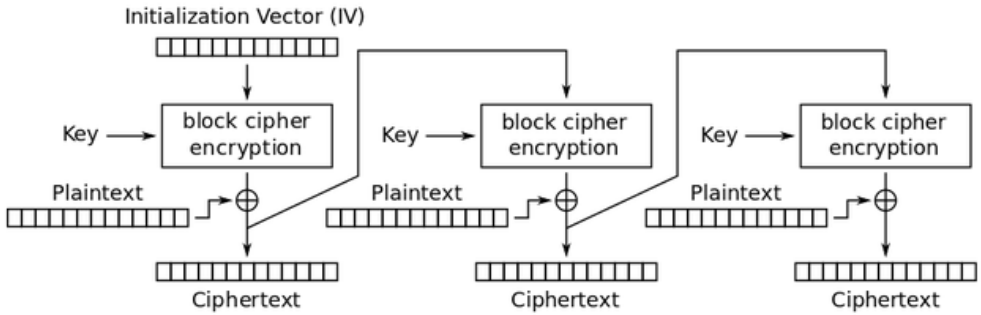
Cipher Block Chaining (CBC) mode encryption



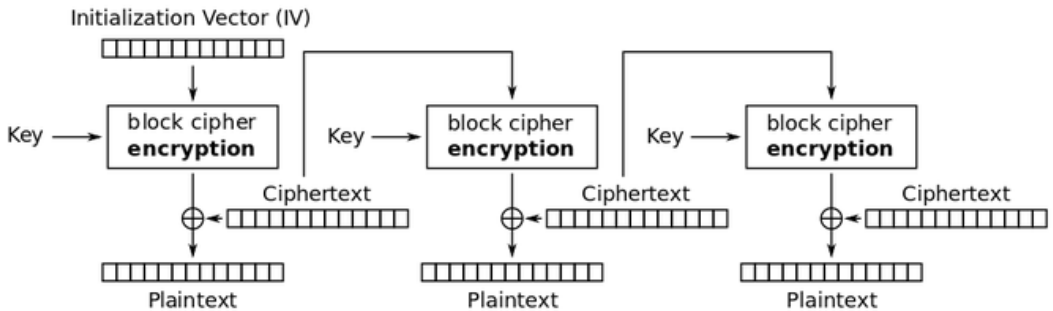
Cipher Block Chaining (CBC) mode decryption

Source of picture : http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation

CFB



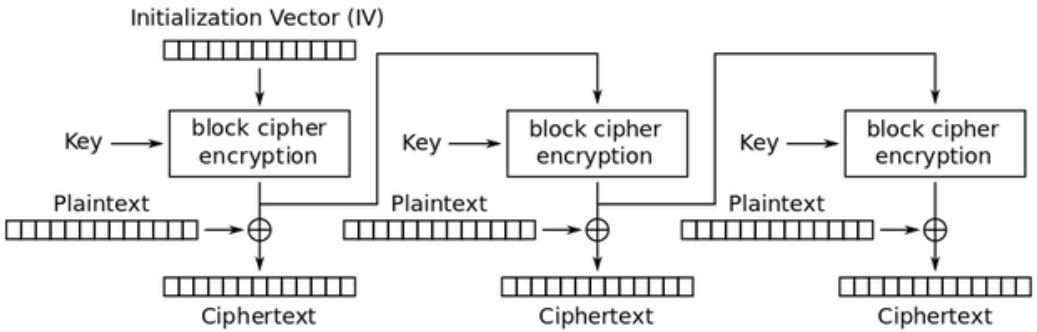
Cipher Feedback (CFB) mode encryption



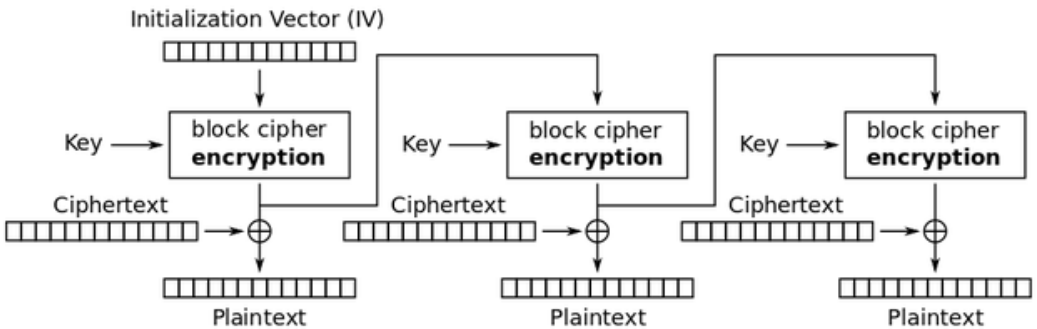
Cipher Feedback (CFB) mode decryption

Source of picture : http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation

OFB



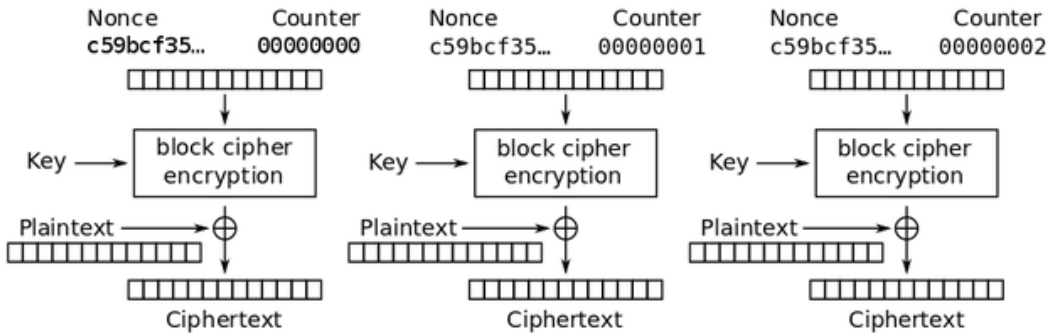
Output Feedback (OFB) mode encryption



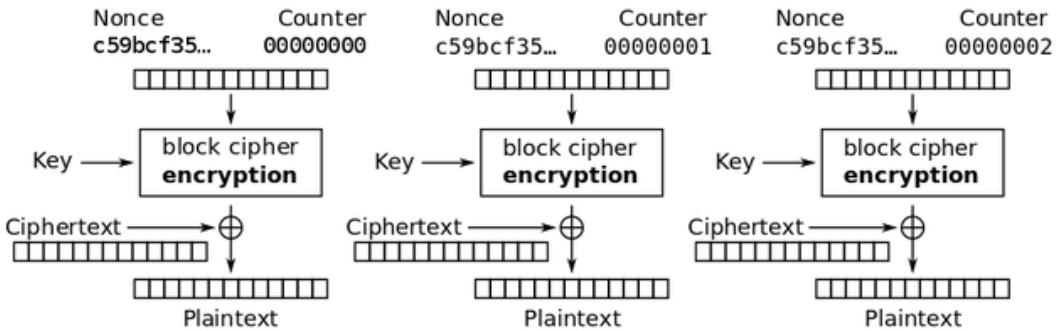
Output Feedback (OFB) mode decryption

Source of picture : http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation

Counter Mode



Counter (CTR) mode encryption



Counter (CTR) mode decryption

Source of picture : http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation

Which Mode of Operation would you Use ?

Question

In your opinion, which mode of operation is the best ?

Modes of Operations Recommendations

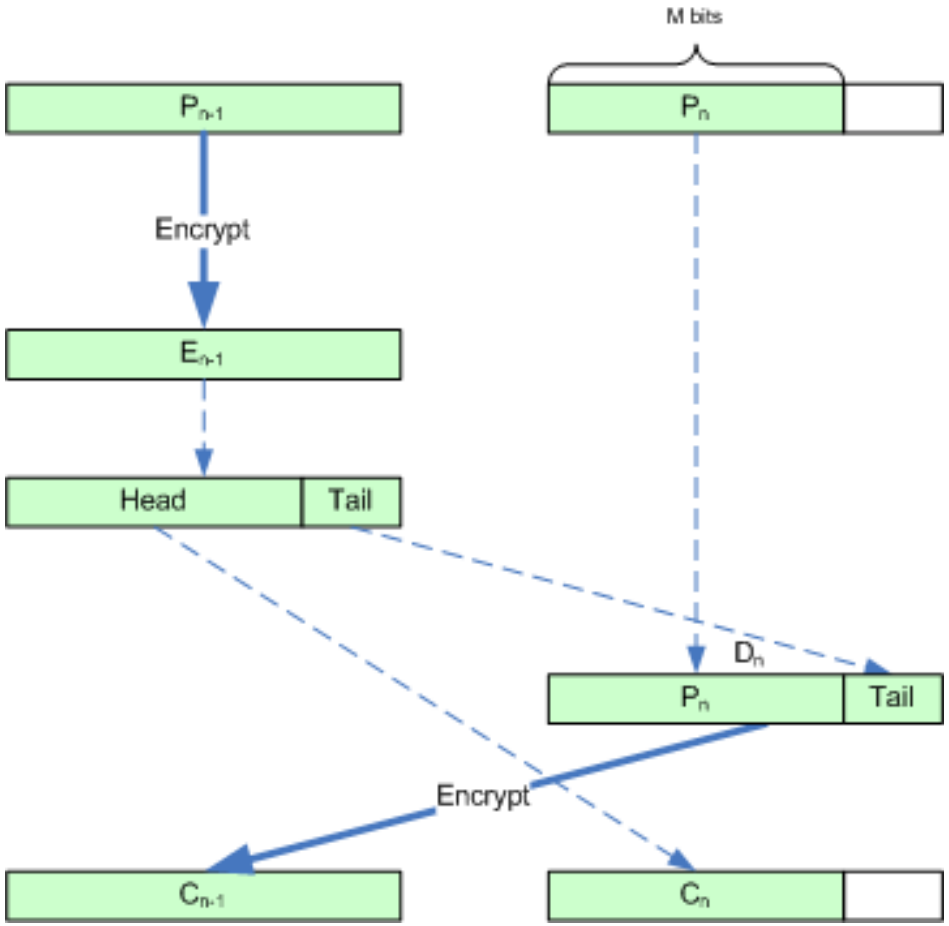
Primitive	Legacy	Future
ECB	x	x
CBC	✓	x
OFB	✓	x
CFB	✓	x
CTR	✓	x
XTS	✓	✓(not for ECRYPT)
EME (patented)	✓	✓
CBC-ESSIV	✓	x
Generic Composition	✓	x
CCM	✓	x
EAX	✓	✓
GCM	✓	✓if used properly
OCB v1.1	✓	✓
ChaCha20 + Poly1305	✓	✓
AES-GCM-SIV	✓	✓

source of recommendations (except the last one) :

Ciphertext-Stealing Mode

- Allows to deal with messages whose length is **not** a multiple of the underlying block cipher's block size, without any expansion of the ciphertext.
- Not standardized, but one can find it implemented in many situations

Ciphertext-Stealing



Source of picture : http://en.wikipedia.org/wiki/Ciphertext_stealing

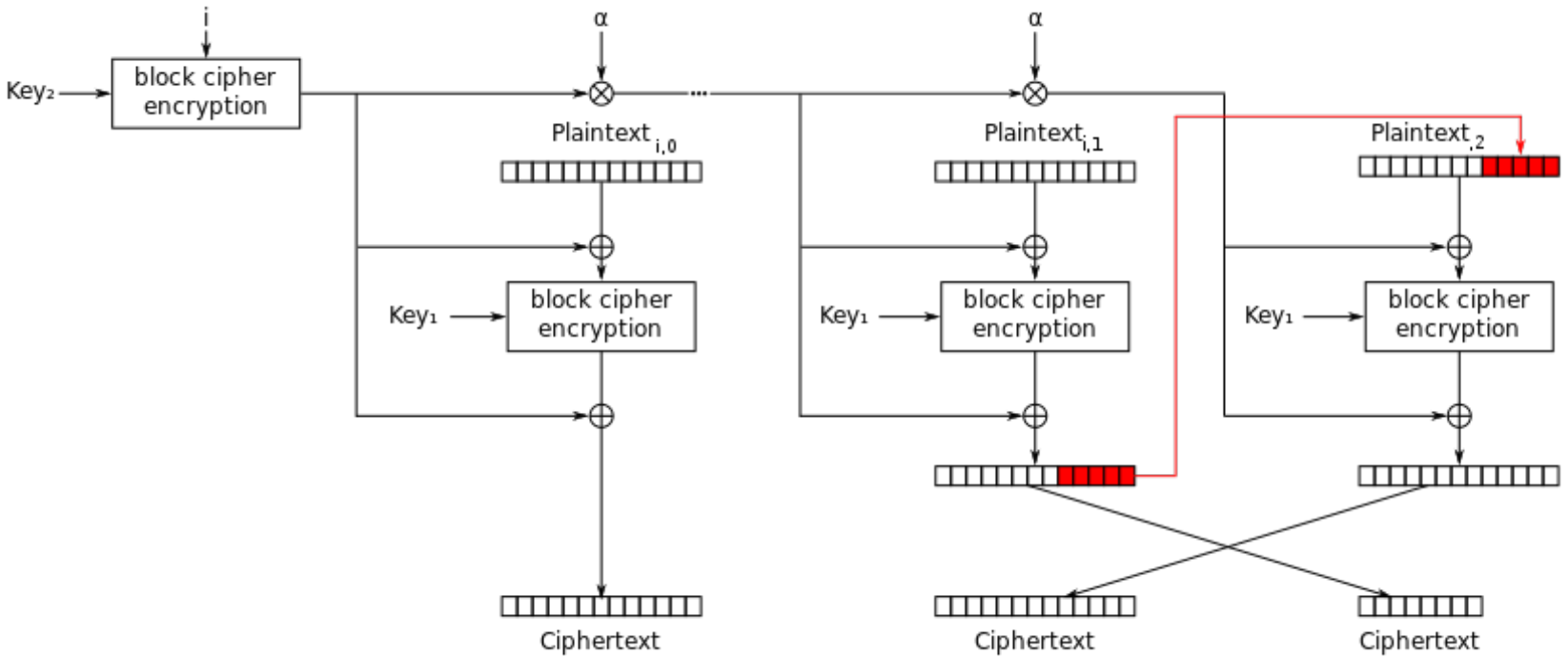
Disk Encryption

- Disk encryption has special requirements :
  - The data on the disk must remain **confidential**
  - Storage and retrieval of data should be **fast**
  - The encryption scheme must **not add overhead**.
- Adversaries can ...
  - ... **read** the raw contents of the disk at any time ;
  - ... request the disk to **encrypt and store arbitrary files** ;
  - ... **modify unused sectors** on the disk and then request their **decryption**.

XTS

- XTS means "XOR-Encrypt-XOR Tweaked-Codebook mode with Ciphertext-Stealing"
- Standardized in NIST SP800-38E.
- Supported by most disk encryption utilities
- Needs to perform multiplications in a Galois field
- Goal : encrypt index j at sector i.
- Requirements : random access with small overhead and no space increase.

XTS



Source of picture : http://en.wikipedia.org/wiki/Disk_encryption_theory

Stream Ciphers

- Stream ciphers are extremely fast symmetric encryption algorithms
- Their security is **less studied** than block ciphers.
- It is usually **preferable** to use a block cipher in CTR mode than a stream cipher (except maybe ChaCha20).

Which Stream Ciphers would you use ?

Primitive
ChaCha20
RC4
E0
A5/1
A5/2
HC-128
Salsa20/20
SOSEMANUK
Grain128a
Grain
Mickey
Trivium
Rabbit

Stream Cipher Recommendations

Primitive	Legacy	Future
ChaCha20	✓	✓
HC-128	✓	✓
Salsa20/20	✓	✓
SOSEMANUK	✓	✓
Grain128a	✓	✓
Grain	✓	x
Mickey 2.0	✓	x
Trivium	✓	x
Rabbit	✓	x
RC4	x	x
A5/1	x	x
A5/2	x	x
E0	x	x

source of recommendations : https://www.ecrypt.eu.org/csa/documents/D5.4-FinalAlgKeySizeProt.pdf

eSTREAM Portofolio

- Cryptographic competition to find a successor to RC4. Ended in 2008.
- Software-oriented :
  - **HC-128** : 128-bit key, 128-bit IV
  - **Rabbit** : 128-bit key, 64-bit IV, existing theoretical distinguisher
  - **Salsa20/12** : 128-bit or 256-bit key, 64-bit nonce, 64-bit. 12 rounds is not recommended for future use. Updated into **Salsa20/20** or **ChaCha20**.
  - **SOSEMANUK** : 128-bit key, 128-bit IV
- Hardware-oriented :
  - **Grain v1** : 80-bit key, 64-bit IV. Too small parameters. Updated into **Grain128a** : 128-bit key and 128-bit IV.
  - **Mickey 2.0** : 80-bit key, 80-bit IV. Too small parameters.
  - **Trivium** : 80-bit key, 80-bit IV. Too small parameters.

ChaCha20

- Designed by Dan Bernstein.
- Most used stream cipher nowadays. Replacement for RC4 in TLS for Google.
- Resistant to timing and cache attacks.
- 128-bit or 256-bit **key**.
- Uses a 64-bit **nonce** and a 64-bit **position counter**.
- **Variant** with 96-bit **nonce** and 32-bit **position counter** (RFC-7539).
- $f(\text{nonce}, \text{counter}, \text{key}) \rightarrow \{0,1\}^{512}$.
- Possibility to encrypt or decrypt a bloc of 512 bits without computing the whole stream.
- Possibility to have a **bigger nonce** with XChaCha20.

ChaCha20 or AES ?

- Both are well-analyzed and well-studied.
- AES is **faster** when the processor has dedicated instructions.
- Otherwise, ChaCha20 is faster.
- ChaCha20 is a stream cipher. You need to protect the integrity of the messages.
- ChaCha20-Poly1305 used in TLS1.3, SSH, ...

Message Authentication Codes

- Message authentication codes are symmetric signature schemes.
- They do **not** provide repudiation.

MACs Recommendations

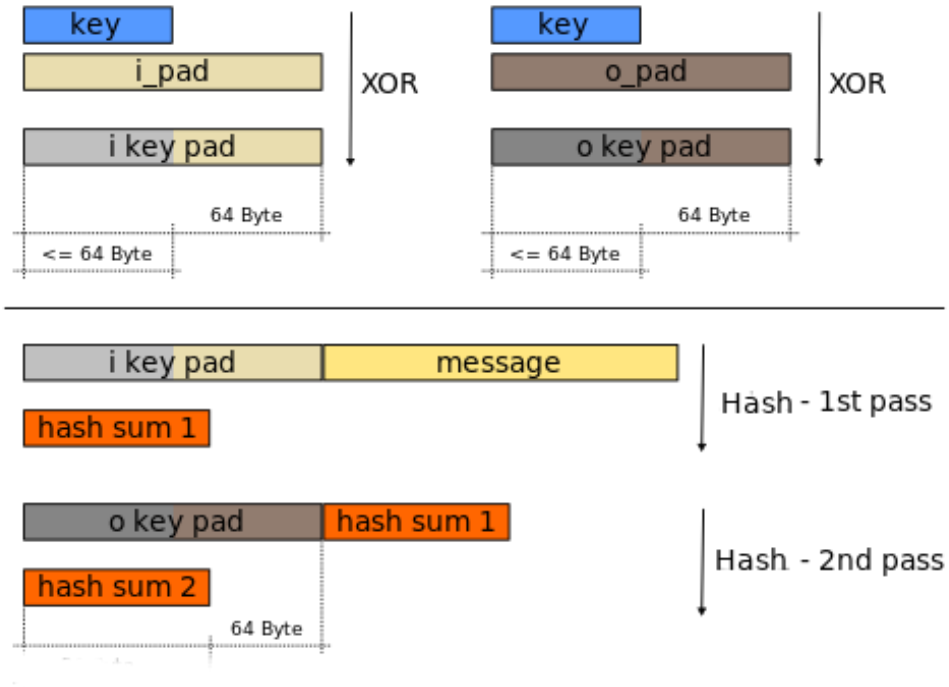
Primitive	Legacy	Future	Comment
HMAC	✓	✓	hash functions
CMAC	✓	✓	Block cipher
EMAC	✓	✓	Block cipher
AMAC	✓	✓	Block cipher
UMAC	✓	✓	Universal hash function
Poly1305	✓	x	When used alone
GMAC	✓	x	Used alone (not in GCM)
CBC-MAC	x	x	

source of recommendations : https://www.ecrypt.eu.org/csa/documents/D5.4-FinalAlgKeySizeProt.pdf

HMAC

- Transforms a hash function in a message authentication code.
- Standardized in RFC 2104 and NIST FIPS PUB 198.
- HMAC-SHA1 and HMAC-MD5 are used in TLS/SSL and IPSec, notably.
- Slow transition towards HMAC-SHA2.

HMAC

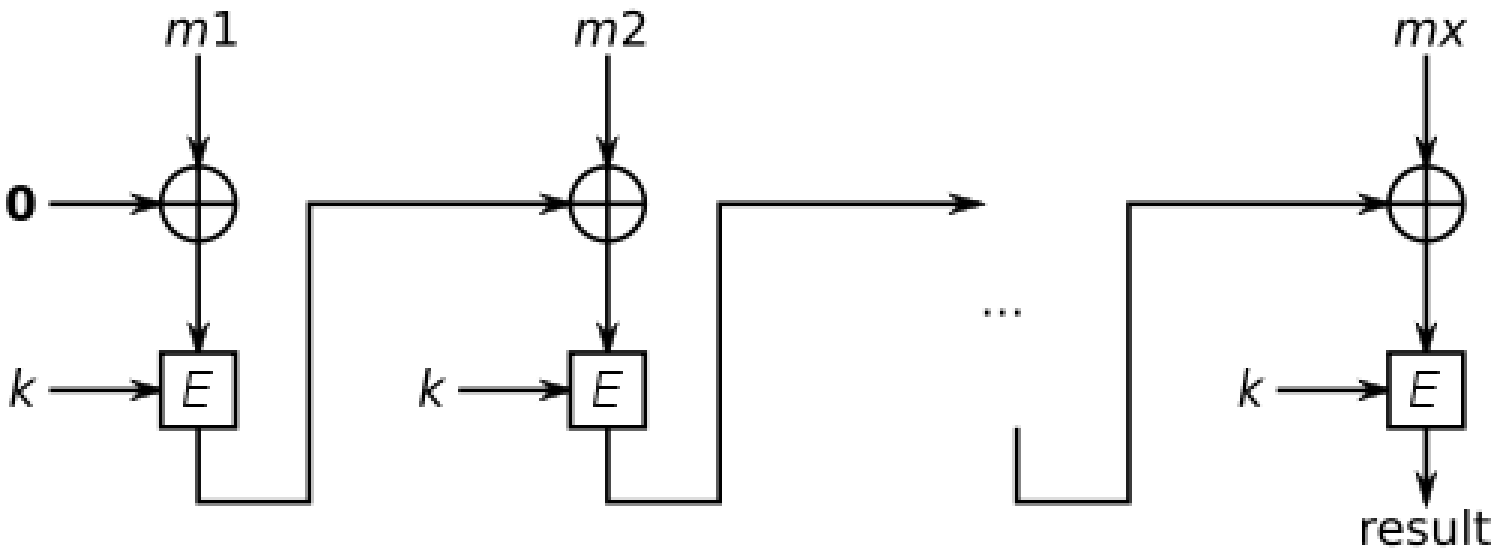


$$\begin{aligned} \text{opad} &= \texttt{0x5C5C5C}\ldots\texttt{5C} \\ \text{ipad} &= \texttt{0x363636}\ldots\texttt{36} \end{aligned}$$

Source of picture : http://en.wikipedia.org/wiki/Hash-based_message_authentication_code

CBC-MAC

- CBC-MAC
  - Based on the CBC encryption mode
  - Uses a block cipher $E_K(.)$
  - Only secure for messages with a **fixed size** !



CBC-MAC Variants

- CBC-MAC is dangerous to use and should not be used.
- **EMAC** encrypts the result with a second secret key.
- The **CMAC standard** was proposed by Black and Rogaway.
- Standardized in RFC 4493 and RFC 4494, as well as in NIST SP800-38B.
- XORs a constant to the last block that depends on the key and truncates the result. The key-dependent constant is $E_K(0)$ shifted by one bit to the left and XOR a constant if there is a carry.
- **AMAC** is another variant of EMAC. Used a lot in banking applications with DES.

Poly1305

- Proposed by Dan Bernstein.
- Often used with Chacha20 or Salsa20.
- Standardized in RFC 7539.
- Requires very simple operations.
- All the integers are represented in **little endian**.

Poly1305 Algorithm (IETF version) – Keys

1. From a **nonce** and a ChaCha20 key, derive a **32-byte key**.
2. Let r be the first 16 bytes and s be the last 16 bytes.
3. Clear the 4 most significant bits of $r[3], r[7], r[11], r[15]$.
4. Clear the 2 least significant bits of $r[4], r[8], r[12]$.
5. We see r as a 16-byte integer.

Poly1305 Algorithm (IETF version) – Computing the MAC

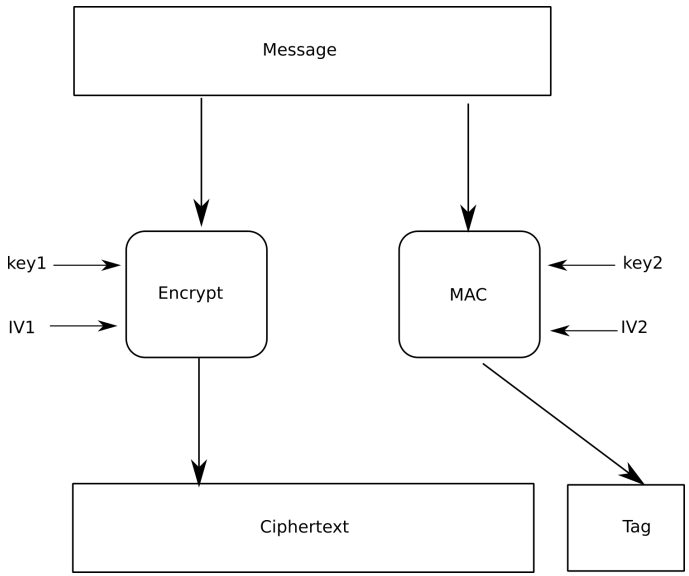
1. Let $p = 2^{130} - 5$ and Acc $= 0$.
2. Divide the message into blocks of 16 bytes. (last one might be shorter)
3. For each message block :
    1. append the byte `0x01` to each block to obtain a 17-byte block. For the shorter block, append further with 0s.
    2. Compute Acc $= (\text{Acc} + \text{block} \cdot r) \bmod p$
4. The final result is $(\text{Acc} + s) \bmod 2^{128}$.

Combining Encryption and MAC

Question

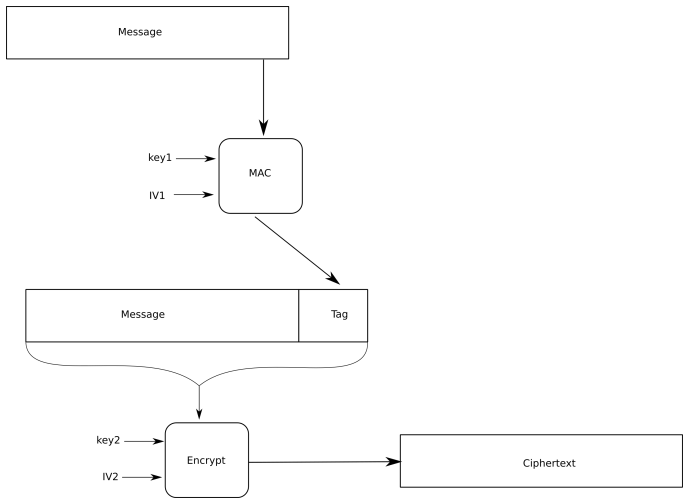
How would you combine a symmetric encryption scheme with a MAC ?

Encrypt-and-MAC



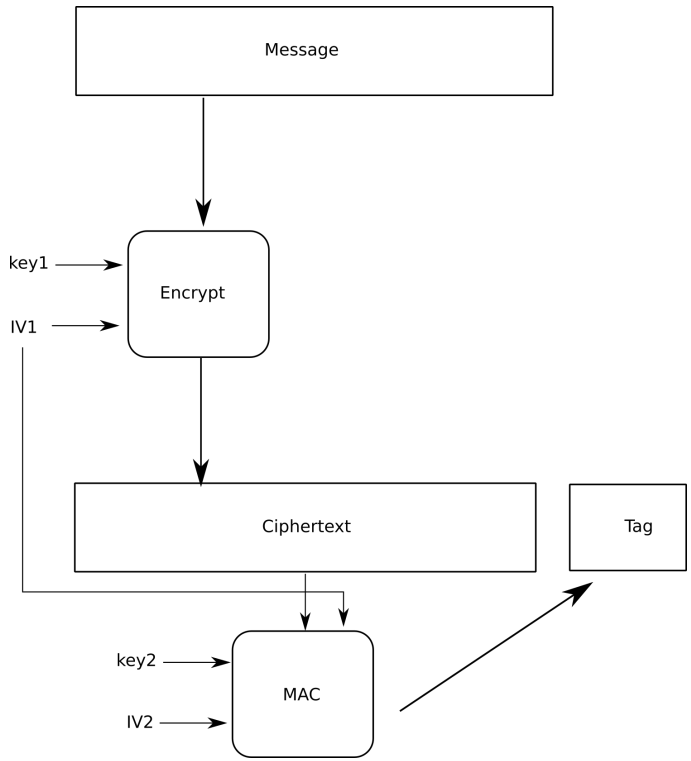
- Globally **not** secure.
- No integrity of the ciphertext : attack on OpenSSH.
- Bad interaction between Encryption and MAC.

MAC-then-Encrypt



- Globally **not** secure.
- Padding oracle attacks on TLS.
- No integrity of the ciphertext.

Encrypt-then-MAC



- Globally the **least risky** solution.
- Protects integrity of the plaintext and the ciphertext.

Same Key for MAC and Encrypt ?

- Globally **risky**.
- Sometimes catastrophic : CBC with CBC-MAC.
- Sometimes no known interaction : AES with SHA2.

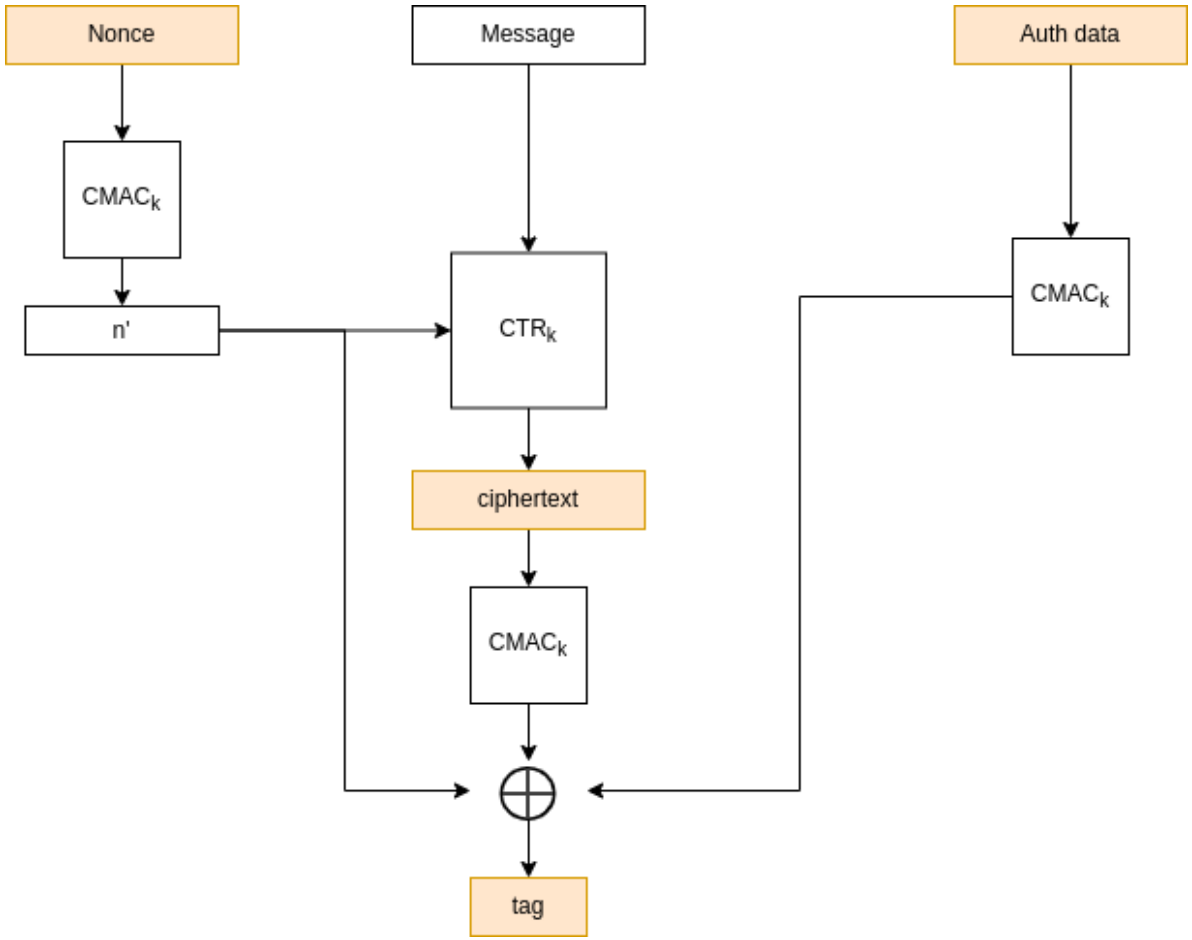
Authenticated Encryption

- **Authenticated encryption** (AE) schemes offer two security properties at the same time : **confidentiality and authenticity**.
- They usually combine a standard mode of operation and a MAC in a clever way.
- Possibility to authenticate data without encrypting it (authenticated data (AD)). AEAD is commonly used for Authenticated encryption with authenticated data.
- **20.02.2019** : announcement of winners of CAESAR competition !
- **Always prefer authenticated encryption when possible !**

CCM

- CCM is a combination of CBC-MAC and the CTR mode.
- CCM uses a block cipher with a 128-bit block size.
- Initialization vectors (IV) must **never** be repeated.
- Standardized in RFC 3610 in combination with AES.
- CCM is used in 802.11i (aka CCMP), IPSec and TLS 1.2.

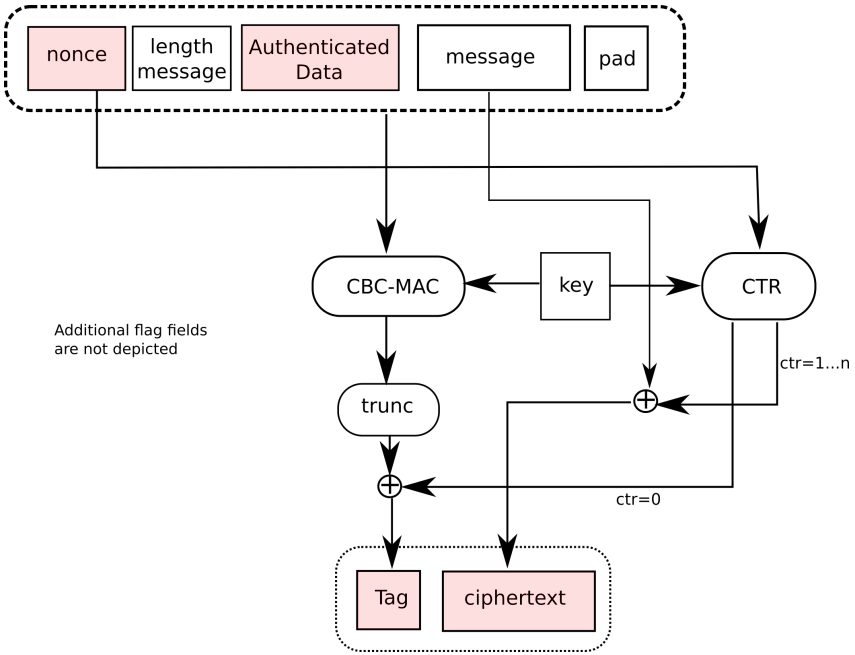
EAX



Question

What are the improvements compared to CCM ?

EAX vs CCM

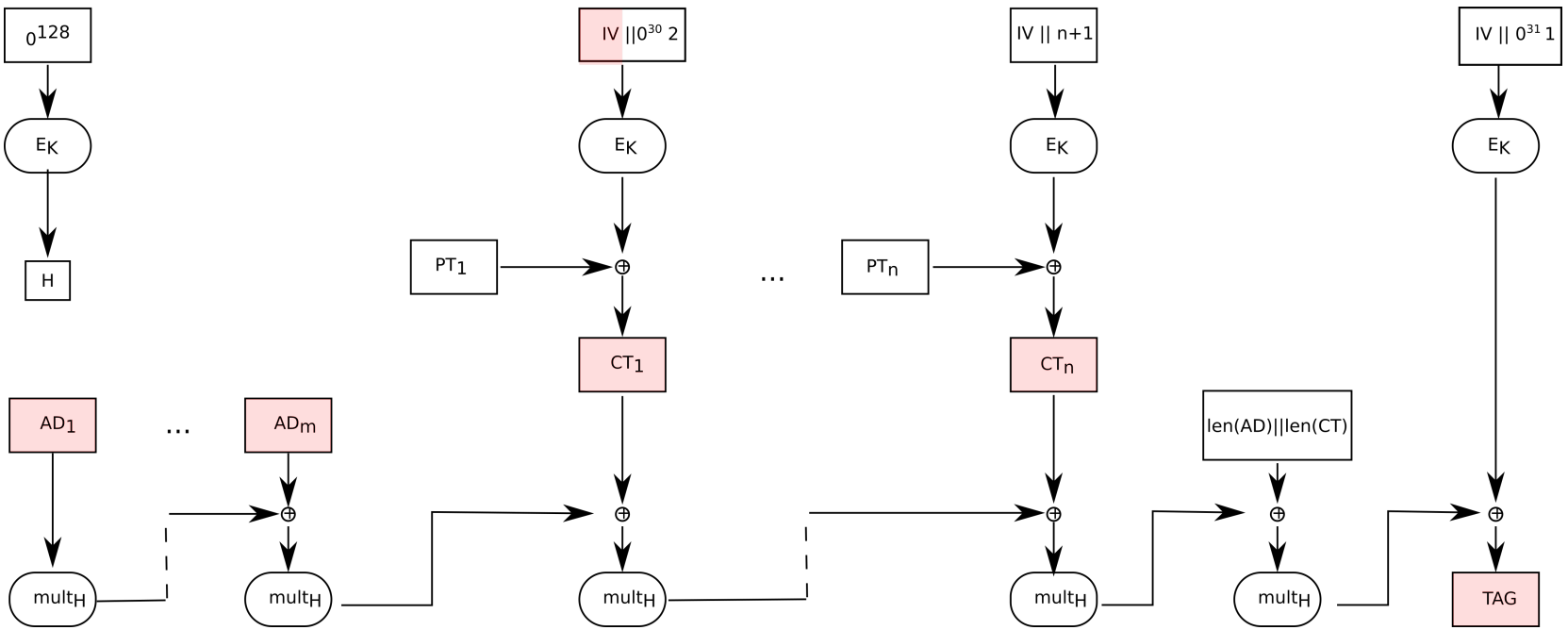


- Strict improvement of CCM.
- Allows to verify the tag before decrypting.
- Can process a stream of data.
- Can pre-process associated data.

GCM

- GCM relies on a block cipher with a 128-bit block size, and accept initialization vectors of any size.
- GCM uses multiplications in GF(2^{128}) constructed as $\mathbb{Z}_2[x]/(x^{128} + x^7 + x^2 + x + 1)$
- Standardized in NIST SP800-38D and is used in IPSec, TLS and SSH, notably.
- The IV (nonce) should **never** be repeated.
- GCM is used in IPSec, TLS, and SSH.
- Better performances than CCM.

GCM


0^{128}
E_K
H
$IV \,||\, 0^{30}\, 2$
$IV \,||\, n{+}1$
$IV \,||\, 0^{31}\, 1$
PT_1
PT_n
CT_1
CT_n
AD_1
AD_m
len(AD)||len(CT)
$mult_H$
TAG

Using GCM

- GCM is **not easy** to use properly.
- The **number of messages** one can encrypt under the same key is **limited**.
- The **size** of a message one can encrypt is **limited**.
- The **IV size** has to be **96 bits**.
- Consult the NIST **special publication 800-38D** for details on what has to be done to obtain a secure implementation.
- Many problems are solved in AES-GCM-SIV (used for instance in BoringSSL by Google).

Birthday Paradox Reminder

- In a space of size d, the probability of having a collision when drawing n values is approximatively

$$1 - e^{-n^2/(2d)} .$$

- This implies that you can find with good probability a collision on ℓ bit hashes with $2^{\frac{\ell}{2}}$ evaluations of the hash function.

GCM Limitations

- The size of **one message** should not be more than $2^{32} - 2$ blocks.
- For GCM with **random IVs**, the **number of messages** encrypted under one key should not be larger than 2^{32}.
- This is obtained with the **birthday paradox** : we want the probability of collision to be $< 2^{-32}$.
- For **deterministic IVs**, the maximum number of messages is 2^{64} (based on how fields are splited in the IV).
- Reusing an IV in GCM breaks **confidentiality** (like for CTR) and **integrity**. One can then authenticate **any message**.

CAESAR Competition

- Started in 2013. About 60 candidates.
- Not managed by a government or a standardization entity.
- Portofolio of winners announced in 2019.
- Winners chosen based on security, analysis quality and performances.

CAESAR Winners

- **Lightweight applications :** Ascon (first choice), ACORN (second choice)
- **High-performance applications :** AEGIS-128 and OCB
- **Defense in depth :** Deoxys-II (first choice), COLM (second choice)

CAESAR : Lightweight Applications

- Fits on small hardware area.
- Fast on 8-bit CPUs.
- Good hardware performances.
- Usually for short messages.

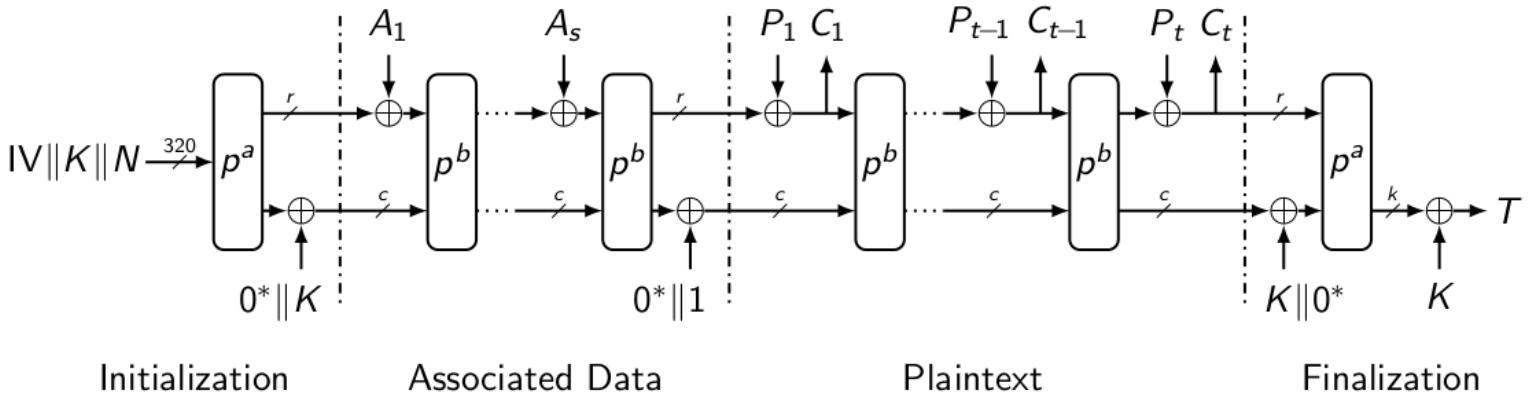
CAESAR : High-Performance Applications

- Efficient on 32-bit/ 64-bit CPU.
- Efficient on dedicated hardware.
- Usually for long messages.

CAESAR : Defense in Depth

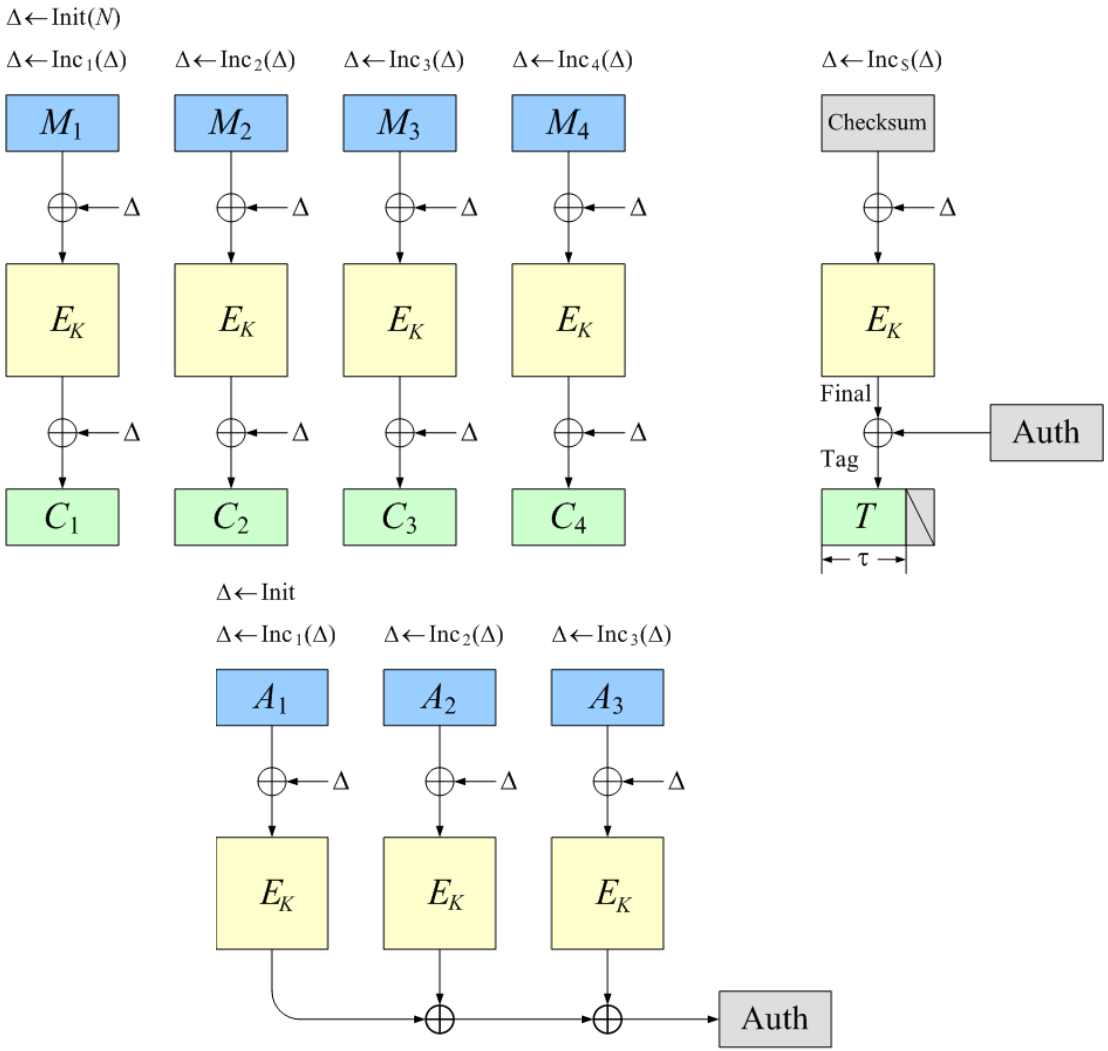
- Protection against **nonce misuse**.
- Limits damage when plaintext are decrypted although not authentic.

ASCON



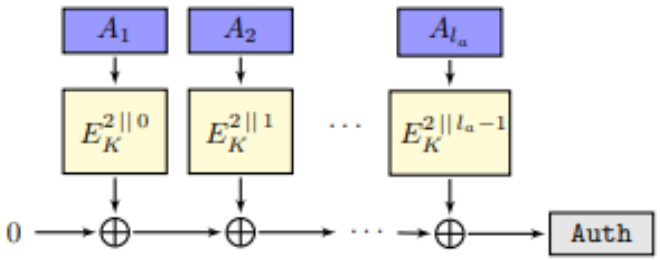
https://ascon.iaik.tugraz.at/specification.html

OCB

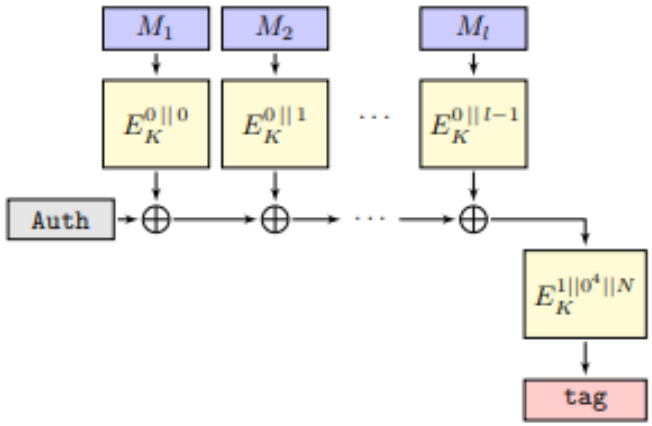


http://web.cs.ucdavis.edu/~rogaway/ocb/ocb-faq.htm

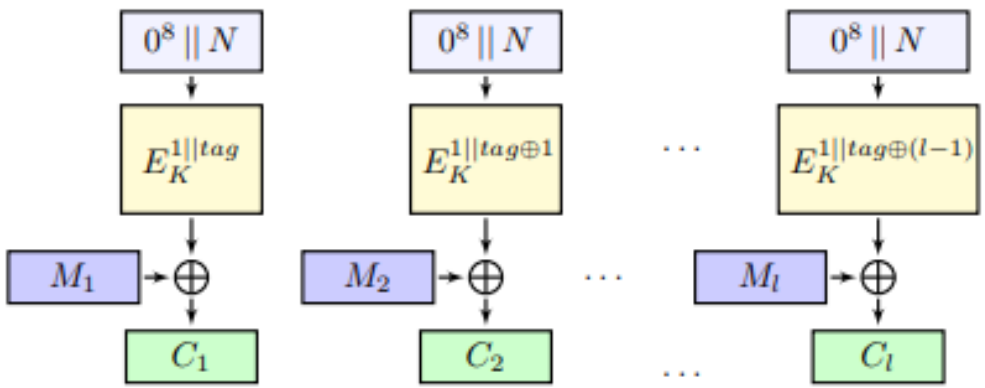
Deoxys-II



(a) Without padding.



(a) Without padding.



(a) Message-length is a multiple of the block size.

https://competitions.cr.yp.to/round3/deoxysv141.pdf

Authenticated Encryption Recommendations

Primitive	Legacy	Future
Generic Composition	✓	x
CCM	✓	x
EAX	✓	✓
GCM	✓	✓if used properly
OCB v1.1	✓	✓
ChaCha20 + Poly1305	✓	✓
AES-GCM-SIV	✓	✓

source of recommendations (except the last one) :

https://www.ecrypt.eu.org/csa/documents/D5.4-FinalAlgKeySizeProt.pdf

Conclusion


AAAAAAAAAAAAAAA
stands for
All Asian, African, American, And Australian Association Against Acronym And Abbreviation Abuse Anonymous
ABBREVIATIONS
Abbreviations.com

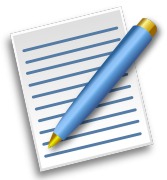
Asymmetric Cryptography Standards

Alexandre Duc

Textbook RSA (Plain RSA)

- Generate two random secret prime numbers p and q.
  Compute $n = pq$.
- Choose a small number e coprime with $\varphi(n) = (p-1)(q-1)$.
- Compute $d = e^{-1} \bmod \varphi(n)$, and erase p, q and $\varphi(n)$.
- **Public key** : (n, e). **Private key** : (n, d).
- **Encryption** : $c = m^e \bmod n$.
- **Decryption** : $m = c^d \bmod n$.

Small Exponent



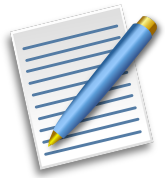
We decide to encrypt an AES 128-bit key with textbook RSA. The AES key will be then used to encrypt data. To make things secure, we decide to select a 2048-bit RSA modulus and $e = 3$.
What attack can you do ?

Small Exponent

Solution

Since the exponent e and the plaintext m are small, m^e (over the reals) is still smaller than the modulus. Hence, the modulus has no effect here. It is, thus, possible to recover easily m from the ciphertext c by computing $\sqrt[3]{c}$. There are simple numerical algorithms doing so (e.g. Newton's method).

Small Exponent and Broadcasting



Suppose that we use textbook RSA with $e = 3$ to send the same message to three different participants.
All three participants have a different RSA modulus, the RSA moduli are 2048-bit long and the message that is sent is also 2048 bit long.
What attack can you perform ?

Small Exponent and Broadcasting

Solution

We will use the Chinese Reminder Theorem (CRT) to increase the space of the message. We have at our disposal three ciphertexts $c_1 = m^e \bmod n_1$, $c_2 = m^e \bmod n_2$, and $c_3 = m^e \bmod n_3$. Using CRT, we obtain a new ciphertext $c = m^e \bmod n_1 n_2 n_3$. The exponent is now much bigger and we can perform the same attack as in the previous question.

Small Exponents

WARNING

Always avoid small exponent, even if the message is formatted.
Coppersmith's attack allows to decrypt when e is small.

A typical (good) choice is $e = 65537$

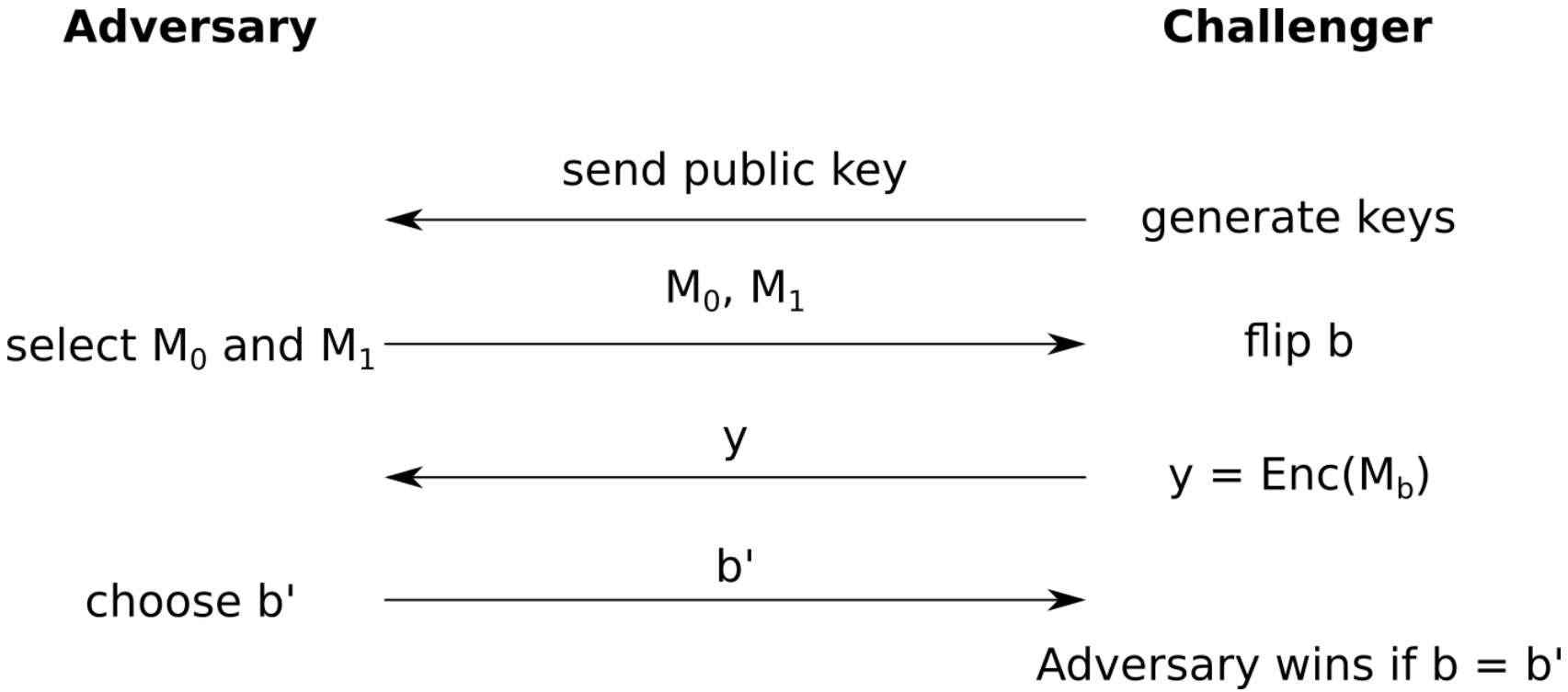
Small Private Keys

Warning

Small private keys are also bad.
Wiener key recovery attack : for $d < \sqrt[4]{N}$.

Don't fix a private key. The inverse of 65537 is extremely likely to **not** be too small.

Indistinguishability under Chosen-Plaintext Attacks (IND-CPA)



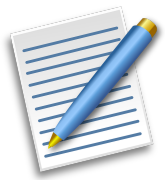
- A cryptosystem is said to be **indistinguishable under chosen-plaintext attack (IND-CPA)** if every *efficient* adversary has only a negligible advantage over random guessing.

IND-CPA Security (formal)

IND-CPA Security

A cryptosystem is IND-CPA secure if $\Pr[\text{win IND-CPA game}] - \frac{1}{2}$ is **negligible** for every PPT (probabilistic polynomial time) adversary.

Textbook RSA : IND-CPA secure ?



Question

Is Textbook RSA IND-CPA secure ?

Textbook RSA : IND-CPA secure ?

Solution

No, it is not. Since we possess the public key, we can try to encrypt both M_0 and M_1 and check which one matches y.

RSA PKCS v1.5

- Older version of the standard RSA encryption and signature padding method.
- Standardized in PKCS#1 v1.5 and RFC 2313.
- Format : `EB = 00 || BT || PS || 00 || D`, where :
  - `BT` is the block type and can be equal to `00`, `01` or `02` ;
  - `PS` is a string of `00`'s (if `BT==00`), or of `FF`'s (if `BT==01`) or of non-zero pseudo-random bytes (if `BT==02`) of at least 8 bytes ;
  - `D` are the data bytes to be encrypted.
- None of the versions of PKCS#1 v1.5 are IND-CPA secure !
- **RSA PKCS#1 v1.5 should be avoided in all new applications !**

Bleichenbacher Attack and Chosen-Ciphertext Security

- Daniel Bleichenbacher, a Swiss cryptographer, has exhibited the first **adaptive chosen-ciphertext** attack against RSA PKCS v1.5 in 1998.
- Chosen-ciphertext security has transformed itself from a theoretical to a practical concern.
- 2017 : ROBOT attack. Attack on SSL/TLS using Bleichenbacher's attack.
- Typical case of **oracle attack**.

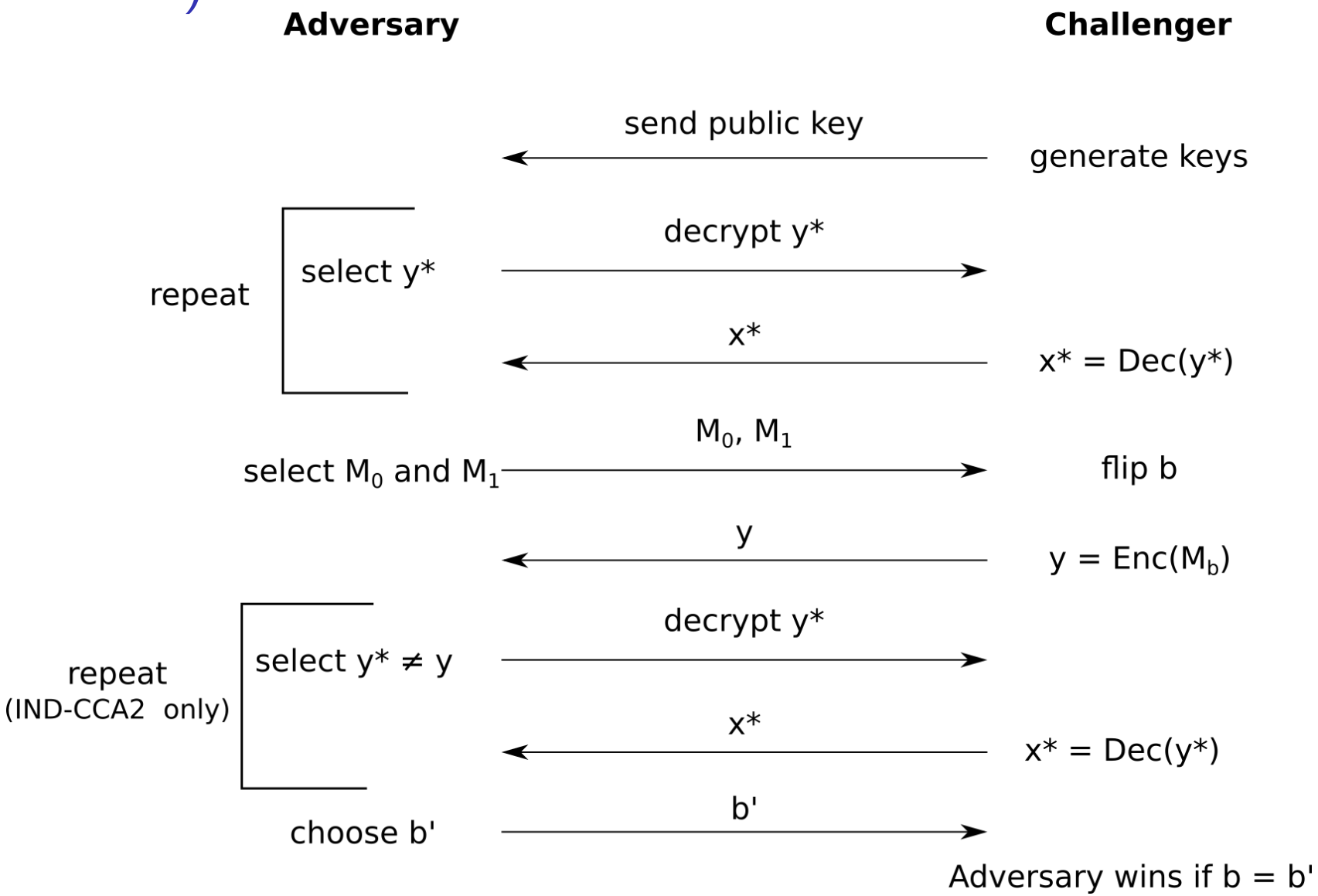
Bleichenbacher Attack : Details

- Let's suppose that `BT = 02` (most common).
- Idea : multiply unknown ciphertext by s^e and use oracle to check if it is valid.
- A valid ciphertext starts with 0x0002. $\rightarrow$ *ms* starts with 0x0002.
- We have $0002\cdots \leq ms < 0003\ldots$.
- With a binary search technique and thousands of queries : can **decrypt ciphertext**.

Warning

Typical **implementation problem** : padding oracle attack.

Indistinguishability under Chosen-Ciphertext Attacks (IND-CCA)



IND-CCA Security (formal)

- Two versions : non-adaptive (IND-CCA) and adaptive (IND-CCA2).

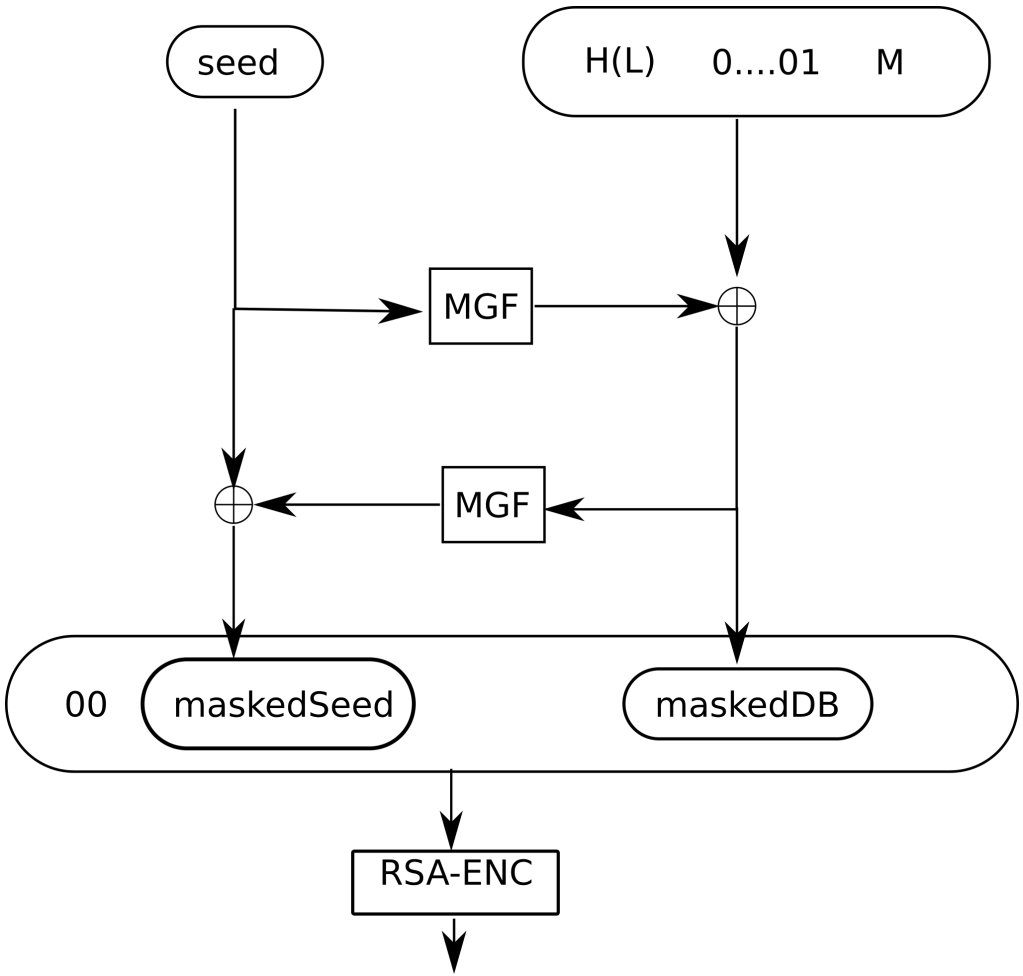
IND-CCA Security

A cryptosystem is IND-CCA secure if Pr[win IND-CCA game] $- \frac{1}{2}$ is **negligible** for every PPT (probabilistic polynomial time) adversary.

RSA-OAEP

- Improved version proposed by Bellare and Rogaway in 1994.
- Padding shown to be IND-CCA2 secure when used with the RSA permutation
- Design goals :
  - Add randomness
  - Prevent partial decryption of ciphertexts : an adversary cannot recover **any part of the plaintext** without inverting the underlying trapdoor one-way function.
- Standardized in PKCS#1 v2.1 and v2.2 as well as in RFC 3447

RSA-OAEP



Manger's Attack on RSA-OAEP

- RSA-OAEP is IND-CCA secure against **black-box** adversaries.
- Manger's attack : Similar to Bleichenbacher's in the idea.
- Two error messages should be **identical**.
- Problem : timings, typos, ...

keylength.com

Method	Date	Symmetric	Factoring Modulus	Discrete Logarithm Key	Discrete Logarithm Group	Elliptic Curve	Hash
[1] Lenstra / Verheul	2022	87	1995 1568	154	1995	164	174
[2] Lenstra Updated	2022	83	1446 1660	166	1446	166	166
[3] ECRYPT	2018 - 2028	128	3072	256	3072	256	256
[4] NIST	2019 - 2030	112	2048	224	2048	224	224
[5] ANSSI	2021 - 2030	128	2048	200	2048	256	256
[6] NSA	-	256	3072	-	-	384	384
[7] RFC3766	-	-	-	-	-	-	-
[8] BSI	2020 - 2022	128	2000	250	2000	250	256

All key sizes are provided in bits. These are the minimal sizes for security.

- There exist many tables.
- It is up to you to choose which ones you trust.

ECRYPT

The goal of ECRYPT-CSA (Coordination & Support Action) is to strengthen European excellence in the area of cryptology. This report [3] on cryptographic algorithms, schemes, keysizes and protocols is a direct descendent of the reports produced by the ECRYPT I and II projects (2004-2012), and the ENISA reports (2013-2014). It provides rather conservative guiding principles, based on current state-of-the-art research, addressing construction of new systems with a long life cycle. This report is aimed to be a reference in the area, focusing on commercial online services that collect, store and process the data.

Protection	Symmetric	Factoring Modulus	Discrete Logarithm Key	Discrete Logarithm Group	Elliptic Curve	Hash
Legacy standard level *Should not be used in new systems*	80	1024	160	1024	160	160
Near term protection *Security for at least ten years (2018-2028)*	128	3072	256	3072	256	256
Long-term protection *Security for thirty to fifty years (2018-2068)*	256	15360	512	15360	512	512

All key sizes are provided in bits. These are the minimal sizes for security.
Click on a value to compare it with other methods.

Recommended algorithms:
Block Ciphers: For near term use, AES-128 and for long term use, AES-256.
Hash Functions: For near term use, SHA-256 and for long term use, SHA-512 and SHA-3 with a 512-bit result.
Public Key Primitive: For near term use, 256-bit elliptic curves, and for long term use 512-bit elliptic curves.

Future algorithms (expected to remain secure in 10-50 year lifetime):
Block Ciphers: AES, Camellia, Serpent
Hash Functions: SHA2 (256, 384, 512, 512/256), SHA3 (256, 384, 512, SHAKE128, SHAKE256), Whirlpool-512, BLAKE (256, 584, 512)
Stream Ciphers: HC-128, Salsa20/20, ChaCha, SNOW 2.0, SNOW 3G, SOSEMANUK, Grain 128a

Elliptic Curve Choice

- Which elliptic curve should I choose ?
- Many different curves : some have multiple different names.
- Three different categories : **Weierstrass, Montgomery, Edwards**.
- Weierstrass curves (seen in class) : $y^2 = x^3 + ax + b$
- Montgomery curves : $By^2 = x^3 + Ax^2 + x$
- Twisted Edwards curves : $ax^2 + y^2 = 1 + dx^2y^2$. **No** point at infinity. The point $(0, 1)$ is the neutral element.

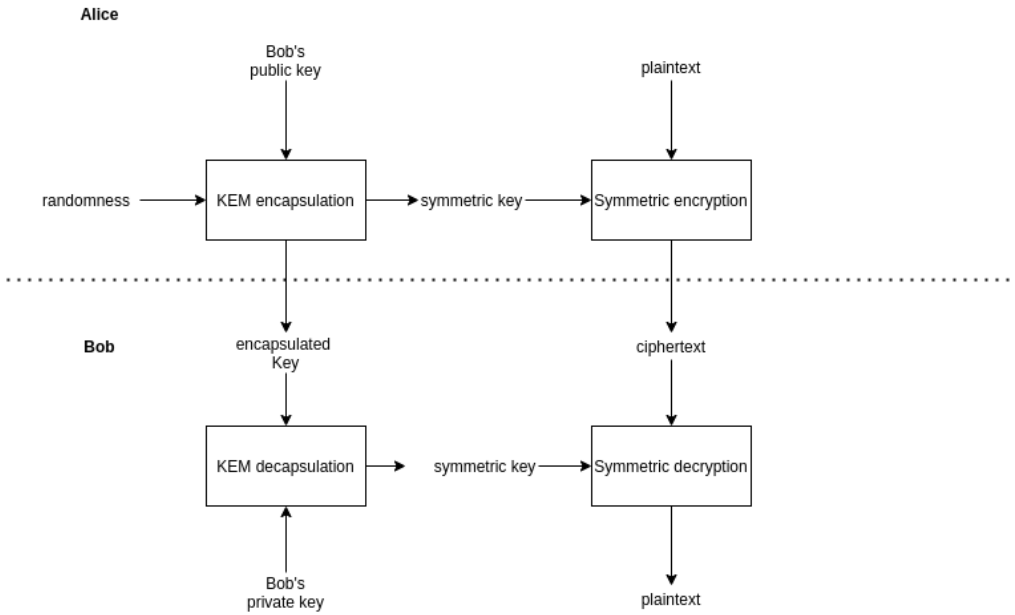
Which Type ?

- Twisted Edwards curves can be mapped to Montgomery form.
- Montgomery curves can be mapped to Weierstrass form.
- **Not** all Weierstrass curves are Montgomery (or Edwards).
- The mappings are described here : https://tools.ietf.org/id/draft-struik-lwip-curve-representations-00.html
- Classical **double-and-add** algorithm is vulnerable to side-channel attacks.
- Solution : **Montgomery ladder** : very efficient for Montgomery (and Edwards) curves.
- Twisted Edwards are needed for **EdDSA**.

Recommendation

Curve	Legacy	Future	Remark
Weierstrass			
W-25519, W-448	✓	✓	From Montgomery curves
P-256, P-384, P512	✓	✓	Hard to implement. Used almost everywhere
P-192, P-224	✓	x	too small
Montgomery			
Curve25519	✓	✓	
Curve448	✓	✓	
Twisted Edwards			
Edwards25519	✓	✓	
Edwards448, E448	✓	✓	
Binary fields	x	x	**Broken**. Favor underlying prime fields

Hybrid Encryption



- **Hybrid encryption** combines an asymmetric encryption algorithm with a symmetric one.
- Two parts : a **Key encapsulation mechanism (KEM)** and a **Data encapsulation mechanism (DEM)**.
- The KEM is asymmetric and encrypts a symmetric key.
- The DEM is a symmetric encryption algorithm.
- **Typical usecase** : encrypted emails.

ECIES

- The **Elliptic Curve Integrated Encryption Scheme (ECIES)** is an integrated encryption scheme that uses the following functions : a key agreement protocol, a key derivation function, a hash function, an symmetric encryption scheme and a MAC.
- Standardized in ANSI X9.63, IEEE 1363a, ISO 18033-2 and SECG SEC 1.
- All the standardized versions show minor differences...

ECIES Setup

- ECIES allows **hybrid encryption** : uses public key crypto to exchange a symmetric key and use symmetric crypto later to send data.
- Public parameters : an elliptic curve E, a point G on E of order n (prime), KDF, symmetric encryption scheme Enc, MAC.
- **Secret key :** $k \in \mathbb{Z}_n$. **Public key :** $K = kG$.

ECIES Encryption(m)

1. Draw $r \in \mathbb{Z}_n^*$ uniformly at random.
2. Let $R = rG$.
3. $(k_E \| k_M) = \mathrm{KDF}(rK)$.
4. $c = \mathrm{Enc}_{k_E}(m)$
5. $\tau = \mathrm{MAC}_{k_M}(c)$.
6. Ciphertext is $R \| c \| \tau$.

ECIES Decryption($R \| c \| \tau$)

Question

How do you decrypt ?

ECIES Decryption($R \| c \| \tau$)

Solution

1. $(k_E \| k_M) = \mathrm{KDF}(kR)$.
2. Verify that $\tau = \mathrm{MAC}_{k_M}(c)$.
3. If correct : $m = \mathrm{Dec}_{k_E}(c)$

RSA-KEM

- **Hybrid encryption** based on RSA.
- See RFC 5990.
- Algorithm :
  1. Draw a random number $z \in \mathbb{Z}_n$.
  2. Let $u = z^e \bmod n$
  3. Use $(k_E \| k_M) = \mathrm{KDF}(z)$ for the data.
  4. $c = \mathrm{Enc}_{k_E}(m)$
  5. $\tau = \mathrm{MAC}_{k_M}(c)$
  6. Ciphertext is (u, c, τ).

Encryption Summary

Warning

Warning with the **implementation** of RSA-OAEP.

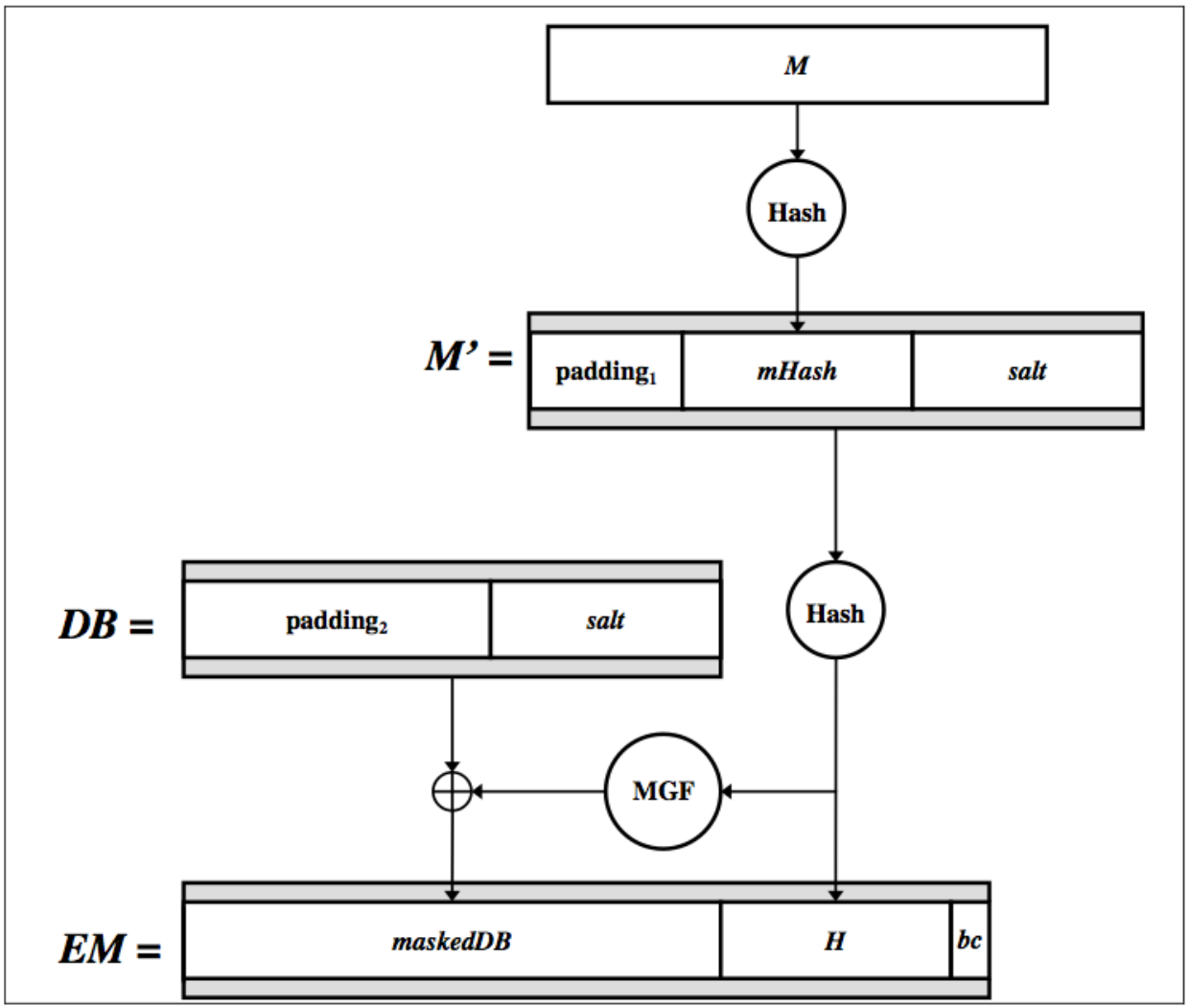
Primitive	Legacy	Future
RSA-OAEP	✓	✓
RSA-KEM	✓	✓
ECIES	✓	✓
RSA-PKCS#1 v1.5	x	x

source of recommendations : https://www.ecrypt.eu.org/csa/documents/D5.4-FinalAlgKeySizeProt.pdf

RSA-PSS

- RSA-PSS uses the Probabilistic Signature Scheme (PSS) proposed by Bellare and Rogaway.
- Standardized in PKCS#1 v2.1 and v2.2 as well as in RFC 8017.
- Relies on randomization and hash functions.

RSA-PSS


M
Hash
M' =
padding1
mHash
salt
DB =
padding2
salt
Hash
MGF
EM =
maskedDB
H
bc

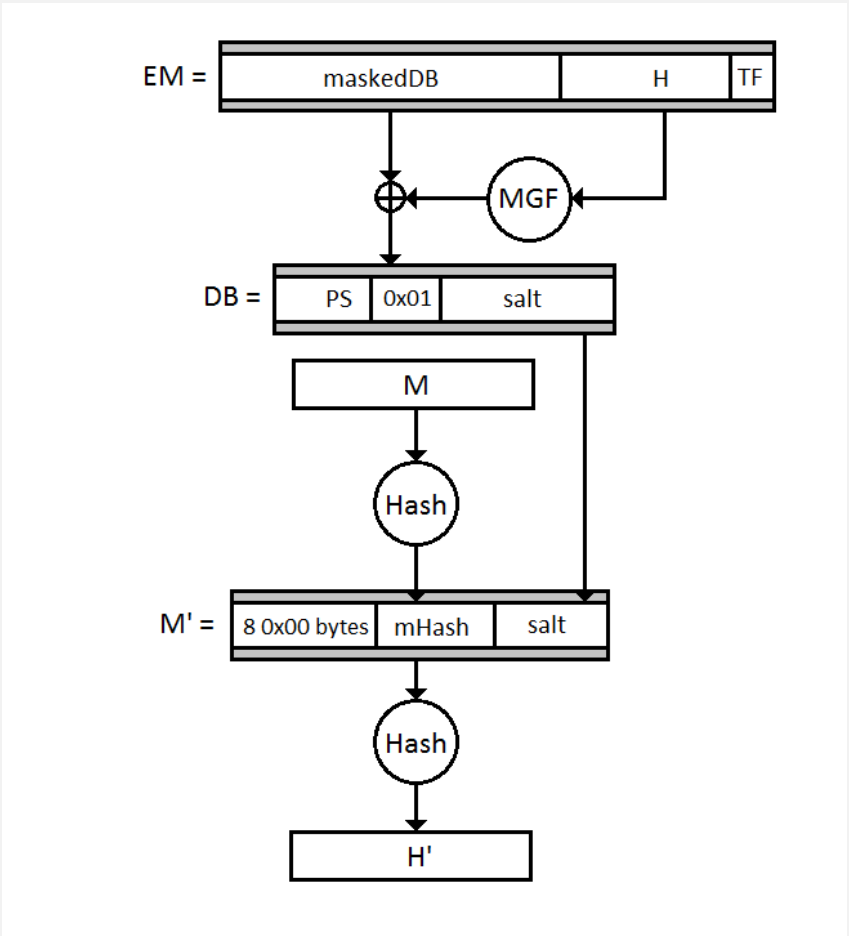
RSA-PSS Verification

Question

How do you verify the signature ?

RSA-PSS Verification

Solution



DSA

- DSA stands for **Digital Signature Algorithm**
- DSA is a variant of El-Gamal signatures and its security relies on the discrete logarithm problem in $\mathbb{Z}_p$.
- DSA is standardized in NIST FIPS 186-4 (known as the "Digital Signature Standard (DSS)") and withdrawn in FIPS 186-5.

DSA : Key Generation

- We work in $\mathbb{Z}_p^*$ with an element $g \in \mathbb{Z}_p^*$ of prime order q.
- Private Key : $a \in \mathbb{Z}_q$, public key : $A = g^a \bmod p$.

DSA : Signature

- We use a cryptographically secure hash function $h : \{0,1\}^* \longrightarrow \{1, \ldots, q-1\}$.
- We generate a uniformly random number $k \in \{1, \ldots, q-1\}$.
- To sign a message m, we compute $r = (g^k \bmod p) \bmod q$ and $s = k^{-1}(h(m) + ar) \bmod q$.
- The signature of m is (r, s) if $r \neq 0$ and $s \neq 0$. Otherwise, restart with a fresh k.

DSA : Verification

- We first **verify** that $0 < r < q$ and $0 < s < q$.
- To verify the signature (r, s) attached to a message m, we verify that $r = \left(g^{h(m)s^{-1}} A^{rs^{-1}} \bmod p\right) \bmod q$.

ECDSA

- ECDSA is a variant of DSA working on elliptic curves.
- It is standardized in NIST FIPS 186-4, X9.62 and SEC2.
- Used in TLS 1.x and SSH
- More and more seen. Will replace DSA.

ECDSA parameters

- We adapt the DSA signature algorithm to elliptic curves (obtaining the ECDSA algorithm).
  1. Choose a cryptographically secure elliptic curve and a point G of order n.
  2. **Private Key** : $a \in \{1, \ldots, n-1\}$.
  3. **Public Key** : $A = aG$

ECDSA signature

To sign a message M :

1. Generate a random, uniform, secret number $k \in \{1, \ldots, n-1\}$.
2. Compute $(x_1, y_1) = kG$.
3. $r = x_1 \bmod n$.
4. $s = \frac{H(M)+ar}{k} \bmod n$
5. The signature is (r, s) if $r \neq 0$ and $s \neq 0$. Otherwise, restart with a fresh k.

ECDSA verification

We verify a signature (r, s) in the following way :

1. We verify that the public key $A \neq \mathcal{O}$, that A is a point on the curve and that $nA = \mathcal{O}$.
2. We verify that r and s are in $[1, n-1]$.
3. We compute $u_1 = \frac{H(M)}{s} \bmod n$ and $u_2 = \frac{r}{s} \bmod n$.
4. We compute $(x_1, y_1) = u_1 G + u_2 A$
5. We verify that $r = x_1 \bmod n$.

Verification Mistakes

- It is essential to verify the values of r and s.
- April 2022 : CVE-2022-21449, vulnerability in Java 15, 16, 17, 18.
- The check is not done for ECDSA : the signature $(0, 0)$ is always valid !
- Why no division by 0 ? $s^{-1} = s^{n-2}$ by the Little Fermat Theorem when $s \neq 0$.
- Efficient way to compute inverses.

DSA and Randomness

Warning

Both DSA and ECDSA are very **vulnerable** to bad randomness for k. One can **recover the private key**.

- One can recover the key when the **randomness** k **is fixed**.
- One can recover the key when the **randomness** k **depends on the previous randomness** : counter, affine function, ...
- If **few bits** of the randomness k leak (two bits is possible) and hundreds of signatures, one can recover the private key.
- With **one byte** only about 20 signatures are required.
- Very interesting combined with a **software bug** : wrong buffer size, buffer overflow, ...

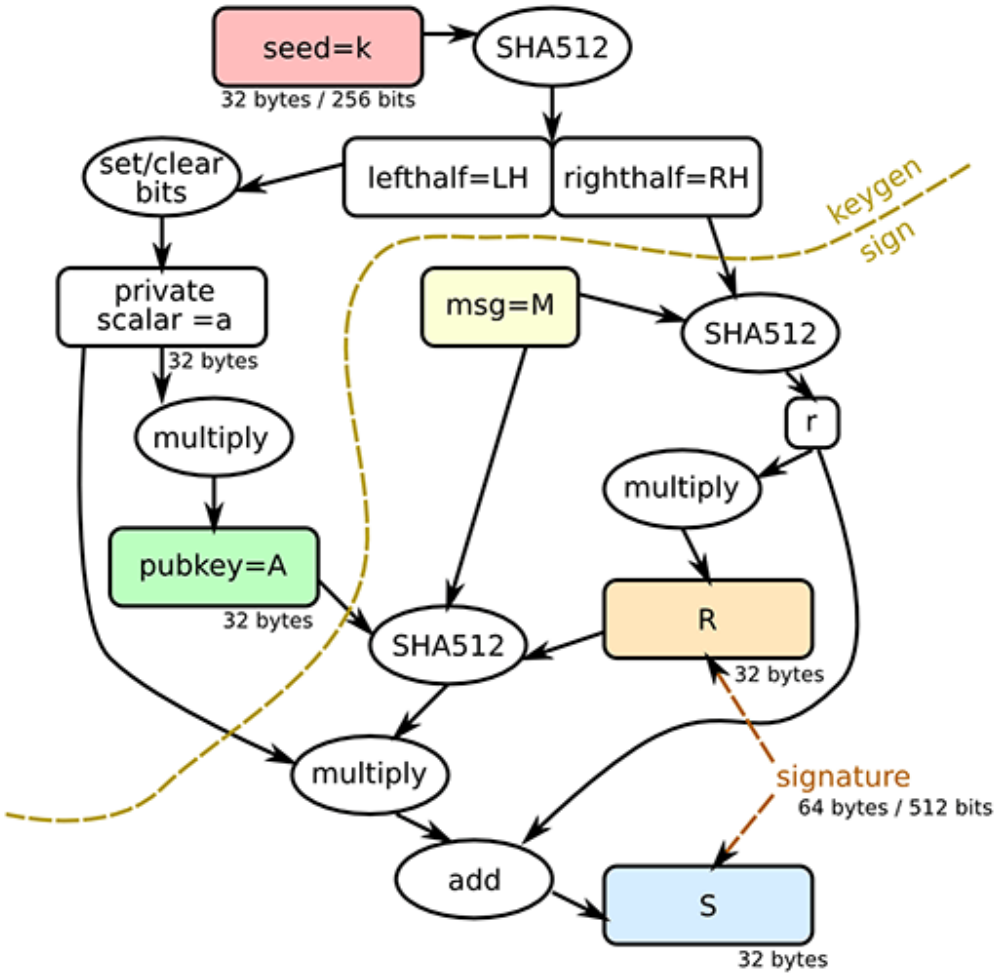
DSA and Security Proof

- There are variants of DSA : **KDSA** and **GDSA**.
- Exist also in elliptic curve variant : **ECKDSA** and **ECGDSA**.
- German and Korean variants.
- The **Korean** variant has a better security proof.
- **Schnorr** signatures are also a better alternative (good proof, simpler equations).
- All these solutions still suffers from randomness problem.

EdDSA

- EdDSA is a **deterministic** signature algorithm.
- Based on **twisted Edward curves**, a special type of elliptic curves with a different equation and different formulas.
- Ed25519 : optimized to be fast on the **x86-64 Nehalem-Westmere** processor family.
- No need of random number generator while signing.
- No branching depending on secrets to avoid side-channel attacks.
- Described in RFC 8032 and introduced in FIPS 186-5.

EdDSA : Global Picture



Source : https://blog.safeheron.com/blog/insights/safeheron-originals/analysis-on-ed25519-use-risks-your-wallet-private-key-can-be-stolen

EdDSA : Parameters and Key Generation

- Let q be a prime number, e.g. $2^{255} - 19$ for Ed25519.
- We are working on an Edward curve E (e.g. Ed25519), with a point B of order ℓ over GF(q).
- The elliptic curve has $2^c\ell$ points.
- H is SHA-512.
- The **private key** k is a random 256-bit string.
- The **public key** is $A = sB$, where $s = H_{\text{msb}(256)}(k)$, i.e., the 256 most significant bits of the hash.

EdDSA : Signature

- The **signature** of a message M is (R, S), with
- $R = rB$, with $r = H(H_{\text{lsb}(256)}(k)\|M)$
- $S = r + H(R\|A\|M)s \bmod \ell$, with $s = H_{\text{msb}(256)}(k)$.
- The **verification** is $2^c SB = 2^c R + 2^c H(R\|A\|M)A$. (Note that the 2^c is not always needed, see RFC).
- **Implementations mistakes** : API allows to provide A that is different from the public key corresponding to k. Allows to recover k.

Making (EC)-DSA Deterministic

- It is possible to make (EC)-DSA deterministic.
- Formalized in RFC 6979.
- The construction derives the nonce from HMAC-DRBG, the private key and the message.
- Gained a lot of popularity.

Minerva Attack

- 2020 : **Minerva attack** : side-channel attack breaking deterministic (EC)-DSA.
- https://minerva.crocs.fi.muni.cz/
- **Timing attack** allows to recover the **nonce bit-length**.
- Sufficient to recover the **private key**.
- Broke many smart cards and cryptographic libraries.

Implementation Point

It is very hard **not** to leak the bit-length. EdDSA seem to avoid this problem because we take SHA hash which is **not** modulo the order of the curve.

Signature Recommendations

Primitive	Legacy	Future	Remark
RSA-PKCS#1 v1.5	✓	x	no proof
RSA-PSS	✓	✓	Big sizes
(EC)-DSA	✓	x	randomness danger
(EC)-GDSA	✓	x	randomness danger
(EC)-KDSA	✓	✓	randomness danger
(EC)-Schnorr	✓	✓	randomness danger
EdDSA	✓	✓	deterministic

source of recommendations (except last) :

https://www.ecrypt.eu.org/csa/documents/D5.4-FinalAlgKeySizeProt.pdf