Key and Random Generation

Nicolas T. Courtois

- **U**niversity **C**ollege **L**ondon

****Part 0 [revision]

Key Sizes

(symmetric encryption +
some asymmetric equivalents)

Key Sizes

What kind of keys can be cracked today:

- For a hacker running a network of PC bots: 2^{64}.
- For a large intelligence agency / excentric billionaire: 2^{80}.
  Requires custom or reconfigurable hardware devices (FPGAs or ASICs).

80-bit and 128-bit security.

- 80-bit level: Breakable, not broken yet
  (or if, they don't advertise it).
- 128-bit level: Unbreakable before we become senile. HUGE GAP !

Fact: There is a lot of space between 2^{80} and 2^{128}.
¼ of million x 1 billion times more.

- in optimistic scenario (Moore's Law works) 128 bit keys will be broken in 2070.
- pessimistic: will take longer.

80-bit and 128-bit security

- 80-bit level equivalents (nearly broken):
  - 160-bits for hashing (SHA-1).
  - 1024-bits for RSA.
  - 160 bits for Elliptic Curves.

- 128-bit level equivalents:
  - 256-bits for hashing (SHA-256).
  - About 4096-bits for RSA (!!!!)
  - 256 bits for Elliptic Curves.

Key Sizes

WinZip is using AES-256 (key size=256 bits)

Is it better than AES-128 ?

Key Sizes - relevance

- Key sizes give an upper bound on security.
- Does not guarantee a lower bound.
- However if the cipher is robust enough, they will NEVER be an attack much faster than 2^n.

Key Sizes - relevance

AES-128:

broken faster than exhaustive search,
may mean broken in year 2065…

Therefore there is no probably no point using 256-bit keys, except for e.g. military secrets where a life span of 120 years can just be mandatory requirement (don't discuss it, just do it…).

Except to protect against future quantum computers.

Key Sizes

Is AES-256 better than of AES-128. ?

<u>The opposite can be argued:</u> (imagined scenario)

AES-128 may never be broken in less than 2^{128}.

AES-256: somebody will publish an attack in 2^{253}. He will become famous for breaking AES, yet nothing will happen. In year 2200 AES-256 will still remain unbreakable.

Besides some people believe that 1) quantum computers cannot work 2) the total number of atoms in the accessible part of the universe is less than 2^{256}. Then the key of 256 bits will never be cracked. Not even in 10 000 years.

Part 1

Secret Key Cryptography

Part 1A

Symmetric Key Generation

Vocabulary

"Real"/Physical Random Number generator:

Comes from fundamental physics, complexity (imprecision of measurement) or intrinsic randomness (quantum mechanics).

Pseudo-random number generator:

Expands a short random string into an infinite sequence of random-looking bits.

Keys vs. Bits

Symmetric Cryptography:

Key should be just random, e.g. 128 bits.

Asymmetric Cryptography:

Keys have some algebraic structure,

- Complex setup.
  - Some parameters can be system-wide (for several users).
  - Other are private. May still be hard to generate.

Hard or Easy ?

Is it easy to generate “secure” random numbers ?

Yes. Many simple methods work.

Is it easy to distinguish the best possible source of randomness from a fake one with a hidden setup/trapdoor ?

Impossible in 99.9999 % of cases.

Question:

Assume that we have a "Real"/Physical Random Number generator + some treatment (to remove imperfections or bias). Should it be used ?

Answer: never. Not secure enough.

Source: many people (e.g. the French government DCSSI in their recommendations for the industry) explicitly rules out this usage.

Reason: can be broken or be disabled by the attacker.

RNG

Must be Open Source (or most of the parts should be)

Entropy Extractors

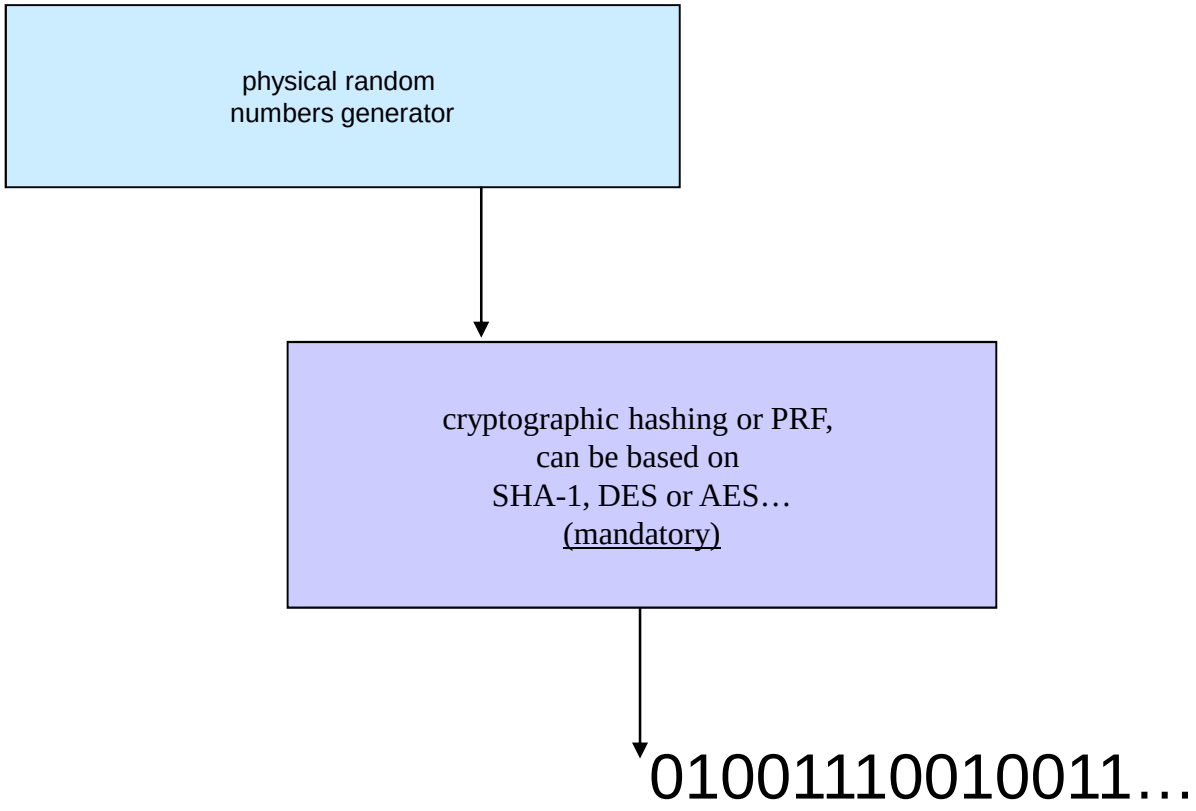
Idea: Use hash function.

Any, even MD5 will do a wonderful job. Does not have to be CR…

Claim: If there is some randomness in the initial generator, the hash function is sure to find it.

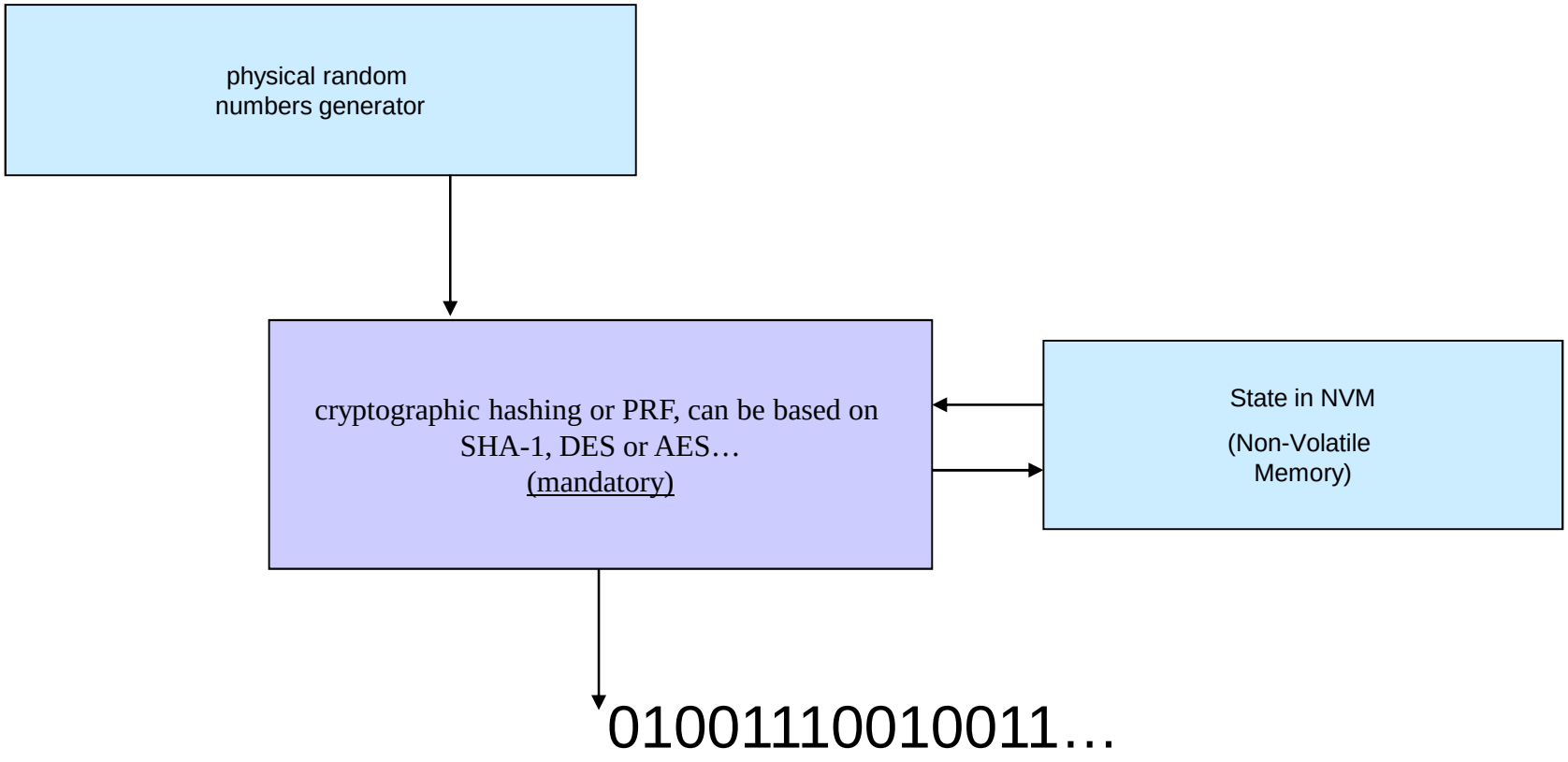
Entropy Extractor Paradigm

better.



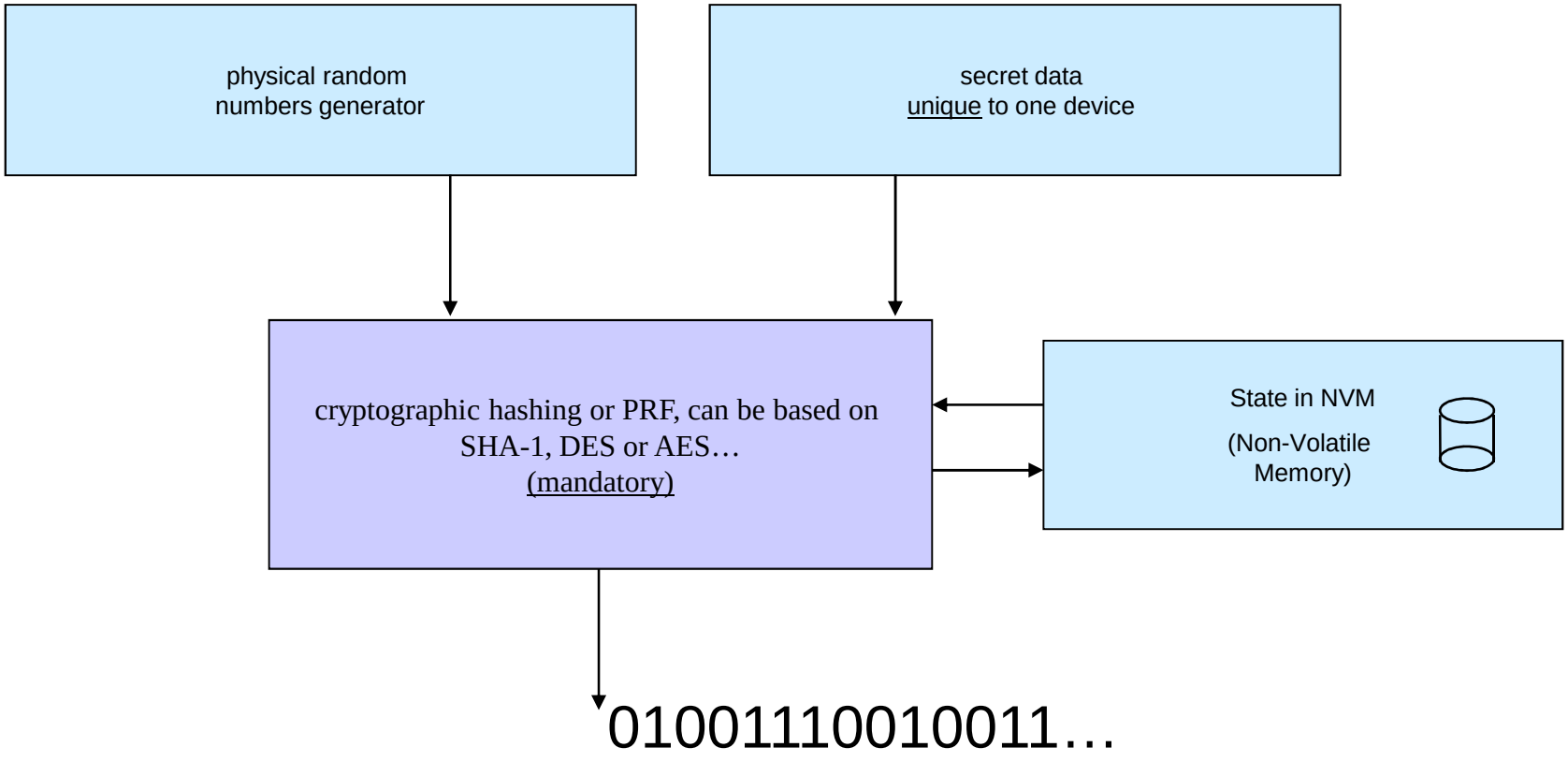
Simple Improvement (hardware)

(prevents “reset” attacks)



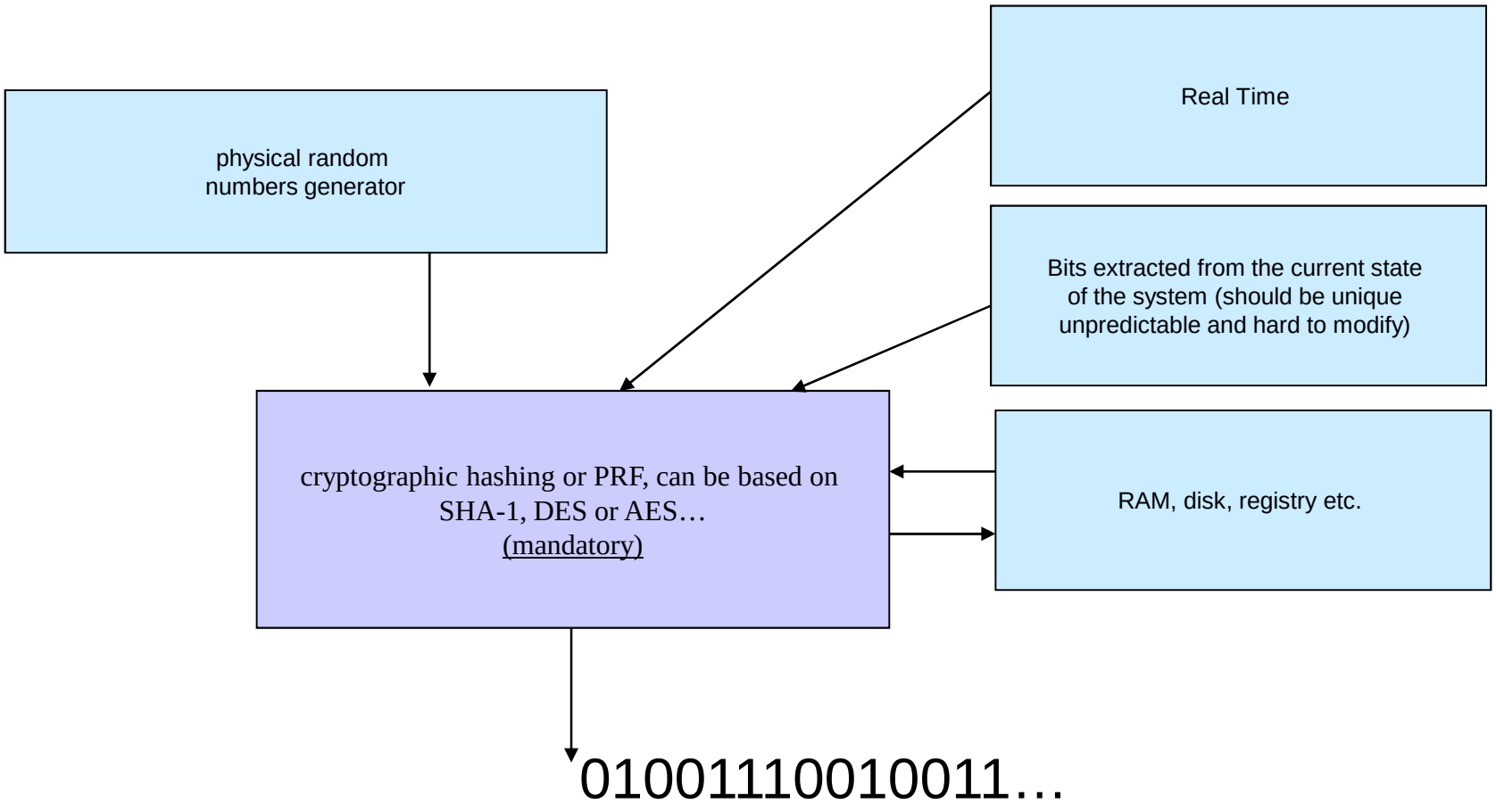
Even better for Hardware

Each device should produce unique numbers.



Simple Improvement - Software Version

(prevents “reset” attacks)



Pseudo-Random Number Generator

Usage:

- where there is no / poor sources of randomness available.
  - Real randomness is in fact expensive (e.g. slow).
- Where we must be able to reproduce the computation exactly (deterministic randomness).

Definition: Expands a fixed-size short random string (called seed) into a much or longer or infinite sequence of "random-looking" bits.

Formal definitions: skip, similar as for OWF.

Inability to distinguish from a really random string of bits.

Pseudo-Random Number Generator

How to Make One ?:

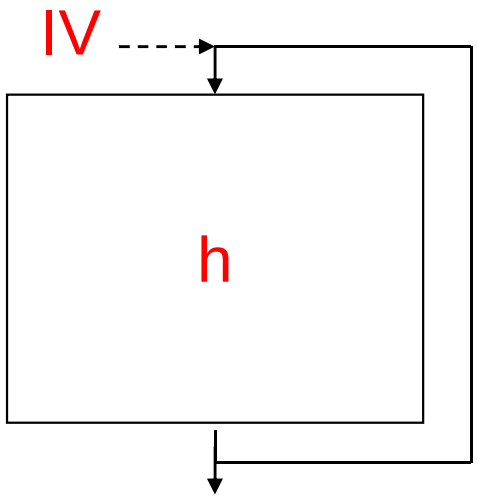
Method 1: Use a synchronous stream cipher such as Snow 2.0.

Pseudo-Random Number Generator

Method 2: use a block cipher or hash function.
Just iterate starting from some initial state.

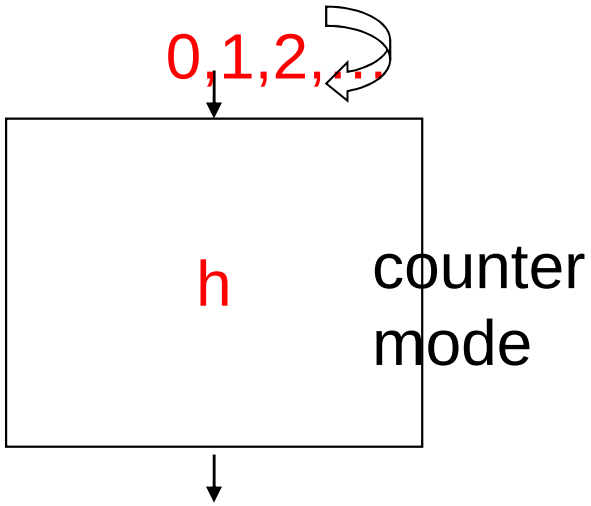
State size >= 160 bits.

Still very good if 128 bits and if the number of bits used is much less than 2^{64}.



Method 3: Counter Mode

Block cipher
OR
hash function
(PRF)



Part 1 B

Nonce Generation

and “Ordinary” Random Numbers
for other applications (like randomising the computations)

The Problem

Vocabulary:

Random nonce, salt, random number,
(also random seed, random coin etc…)

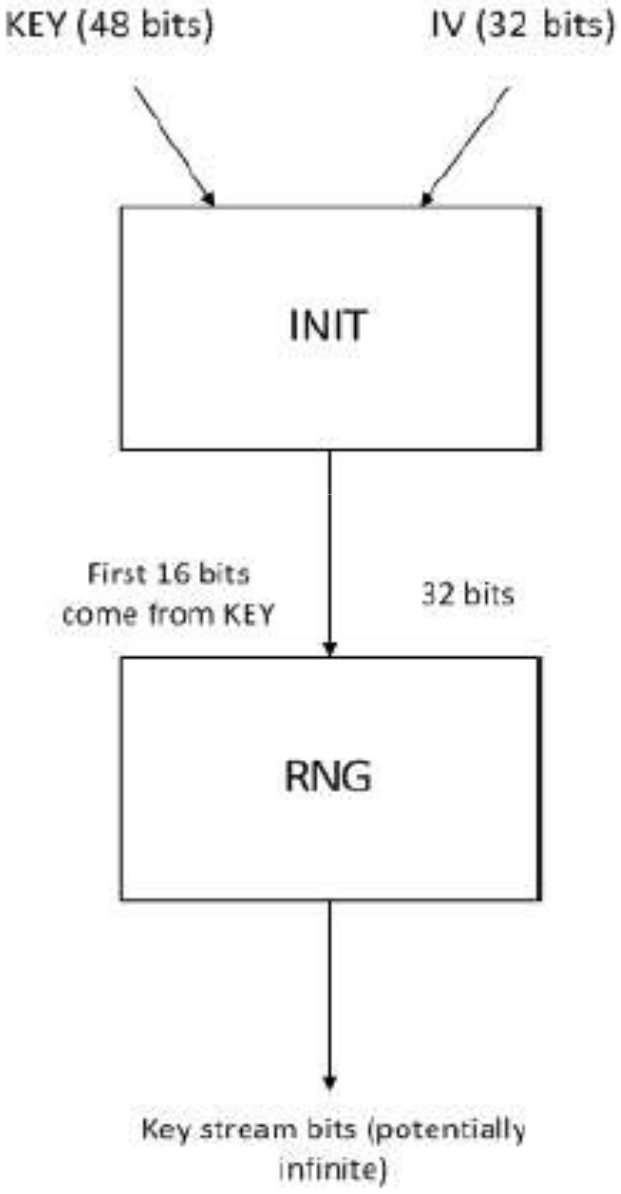
Applications:

Any type of encryption,

- It is usually required that encryption is probabilistic…

Stream Encryption

Example:

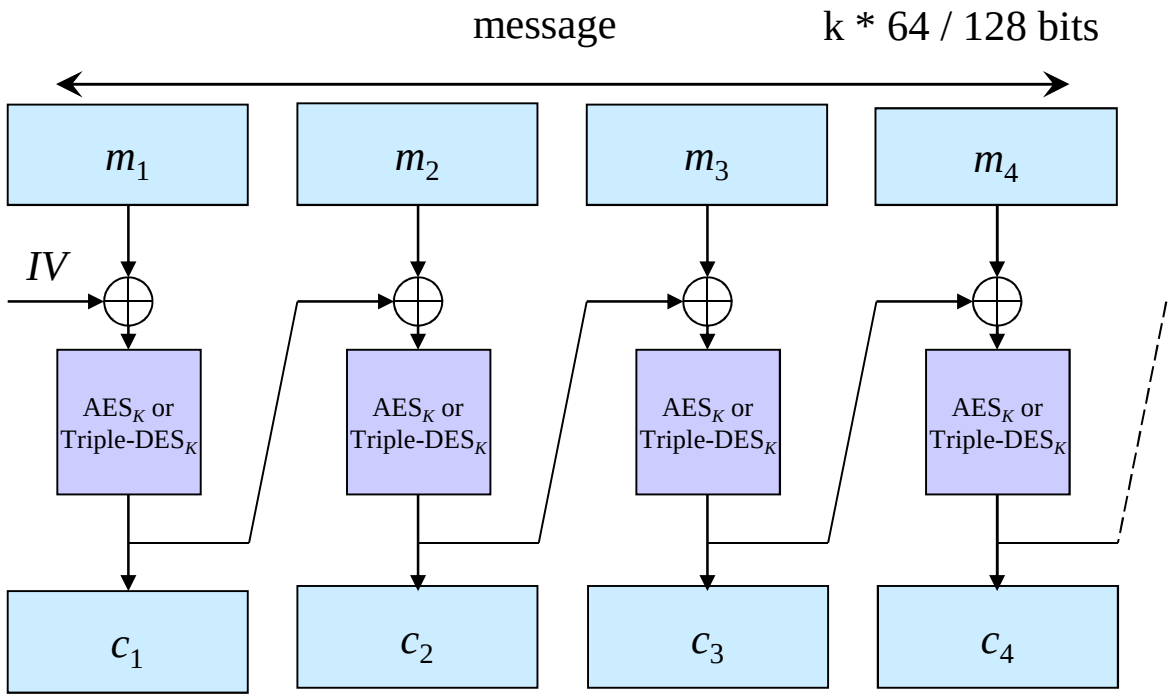


Any "Good" Block Encryption Method

"Must be" probabilistic – requires a random Initialization Value (IV).

Example:

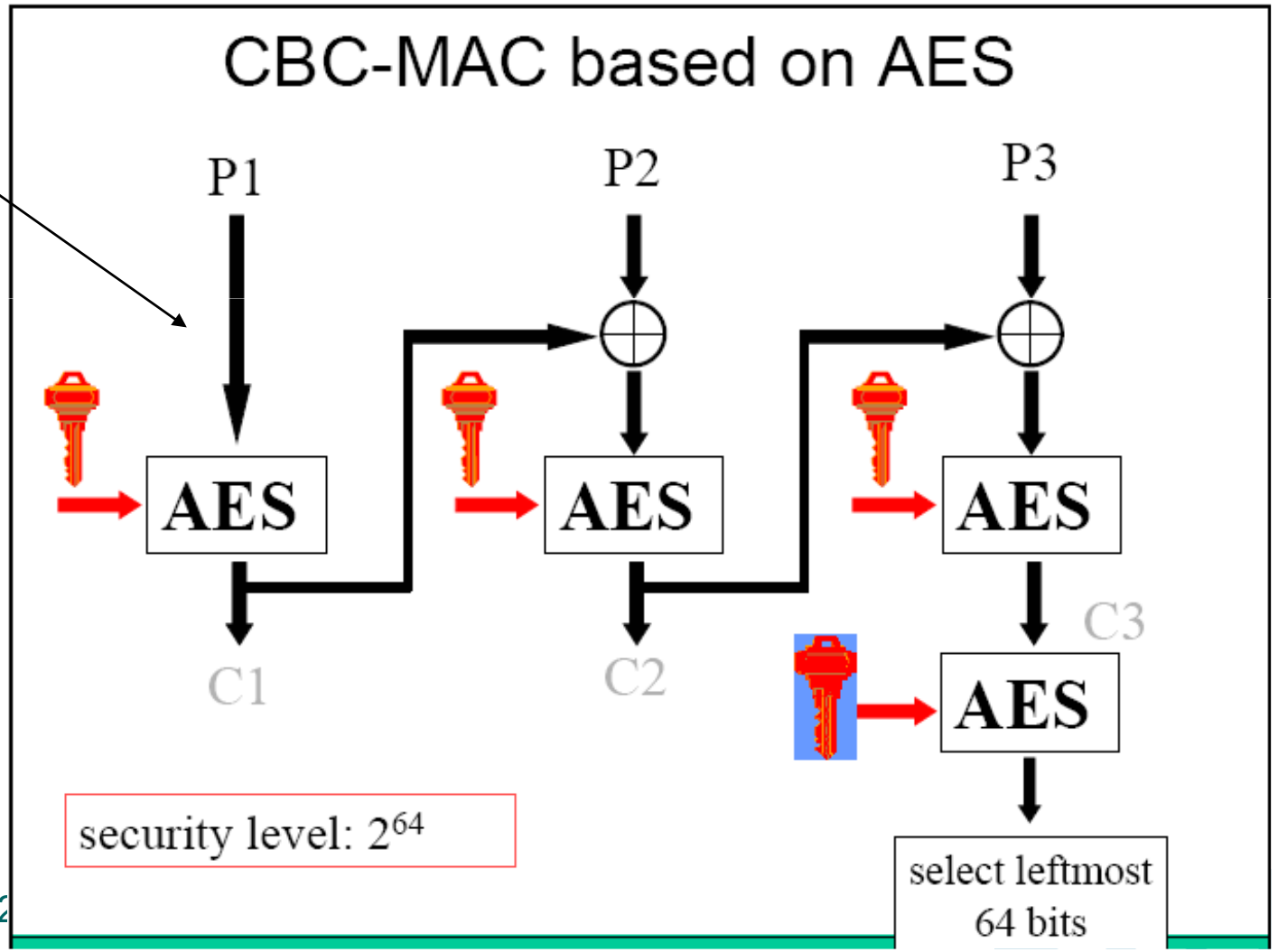
CBC mode:



BEWARE: Message Authentication (MACs)

Should **NOT** use IVs.

Example:
IV=0.



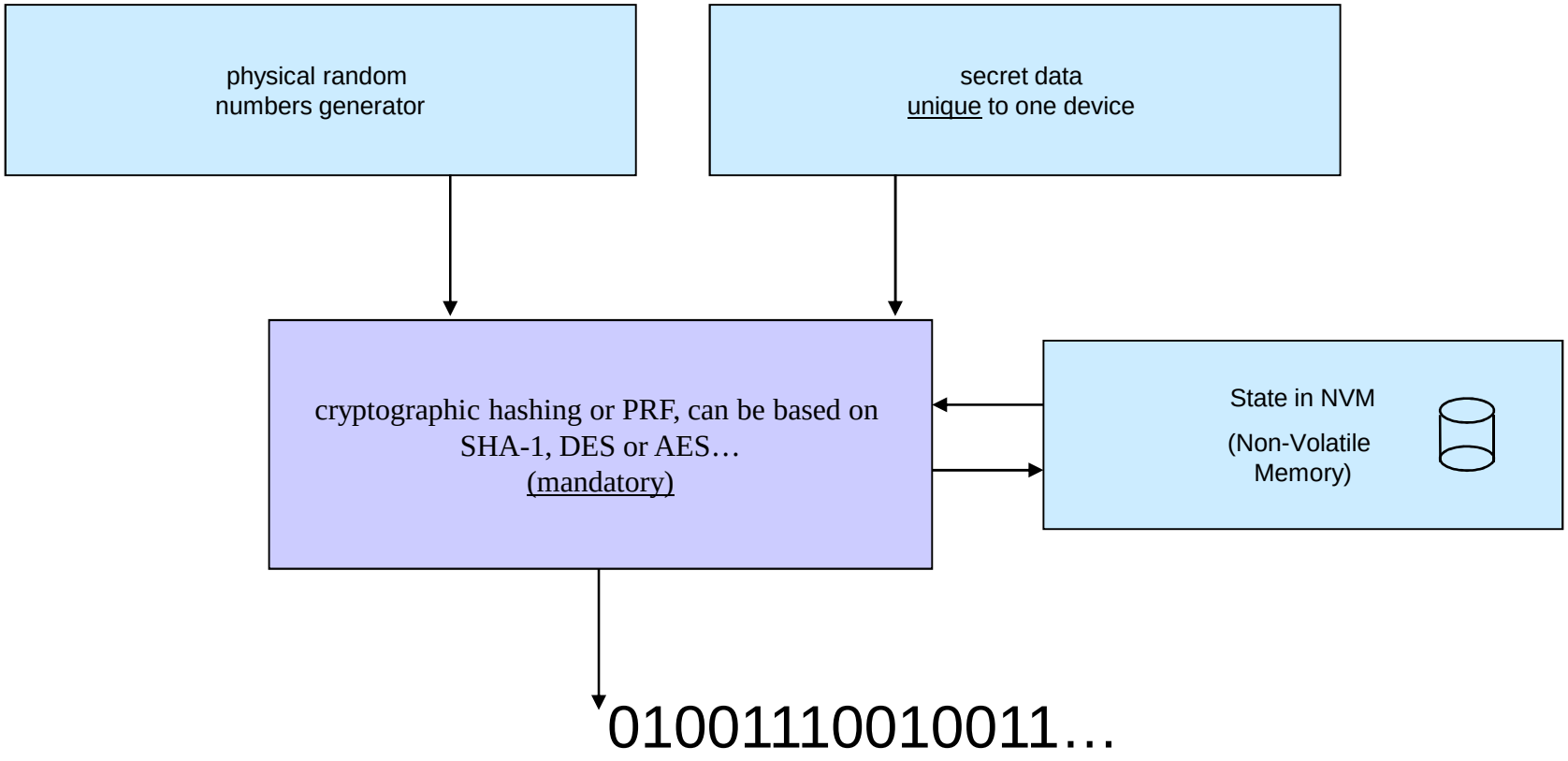
Secure Nonce Generation

Goals:

1. Random-looking.
   - Does NOT have to be random, but
     - informally:
       there should be no way to “simply” relate the data, but there can be a complex relationship involving cryptographically strong functions…
2. Public [or known to at least several parties].
3. In practice nonces should NEVER repeat.
   - This goal is not easy to achieve.
4. Robustness/survivability etc..
   - The system should still work in presence of some hardware problems [faults, reset, ettc…]

Ingredients

The same, BUT the requirements are less strict, the generator can be much faster…



Part 2

Public Key Cryptography

Part 2.1.

RSA Key Generation

RSA Key Generation

- Requires to generate 2 large prime numbers. E.g. 1024 / 2 bits each.
- Not an easy task, and slow. Many smart cards needs tens of seconds for this. Not so serious because done only once in their lifetime.

For example, on Infineon futuristic 32-bit 0.13 um chip SLE88CFX4002P, it takes 3.5 seconds on average to generate a 2048-bit key with the maximum clock speed of 66 MHz. According to Infineon, SLE88CFX4002P is "the most sophisticated smart card microcontroller" on the market.

Random Primes

Method:

- Choose a Number at random.
- Check if it is a prime. (isprime(n) in Maple).
- Try again…

This gives you a guarantee of prime numbers that have no bias or special structure.

Requires many tries…

Theorem: The probability that n is a prime is about $1/\log_e n$.

Remark: Can do much less tries by "sieving":
one only needs to check integers that are congruent to 012345 mod 6. Etc.

Random Primes

Question: [important because randomness is an expensive resource (!):

- How much random bits do we need to generate a random prime on 1024 bits ?

Answer: about 1024 bits only.

Method:

- Choose a random number x on 1024 bits.
- Pick the smallest prime p >= x.

Remains to solve: primality testing.

****** Beware

Our “good” method does NOT give the same random distribution.

Primality Testing

Many methods:

- Miller-Rabin
- AKS method
- ECPP method
- Many many other - see http://cr.yp.to/primetests.html

Miller-Rabin, fast and practical

Let $n-1=2^s * d$, where d odd

Pick a random $a \leq n-1$

Theorem:

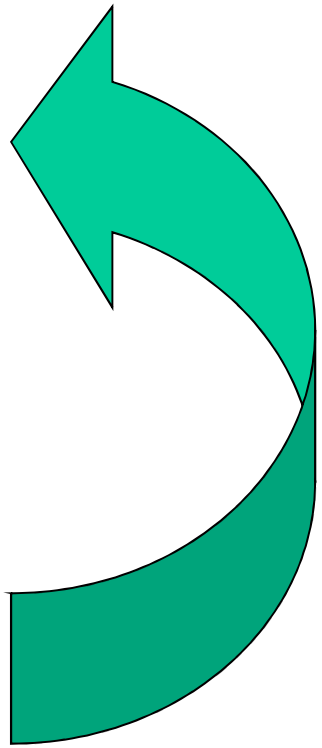
If $a^d \neq 1$ and

$a^{(2\text{^}r*d)} \neq -1$ for all $0 \leq r \leq s-1$

then n is composite.

Otherwise maybe prime,

repeat many >80 times to make sure.



Primality in Strict Polynomial Time

In 2002 Agrawal, Kayal and Saxena [AKS] have shown that

Prime $\in$ P.

Means to check if a number is a prime takes polynomial time.

AKS method

[AKS] P time method.

Based on Fermat's Little Theorem.

```
Input: Integer n > 1

if (n is has the form ab with b > 1) then output COMPOSITE

r := 2
while (r < n) {
   if (gcd(n,r) is not 1) then output COMPOSITE
   if (r is prime greater than 2) then {
      let q be the largest factor of r-1
      if (q > 4sqrt(r)log n) and (n(r-1)/q is not 1 (mod r)) then break
   }
   r := r+1
}

for a = 1 to 2sqrt(r)log n {
   if ( (x-a)n is not (xn-a) (mod xr-1,n) ) then output COMPOSITE
}

output PRIME;
```

AKS: Prime $\in$ P

This result was a breakthrough in number theory.

So far however, their method is under development and is still slower than previously known methods (!).

The are other methods are also polynomial in practice, and faster, but.. What's wrong?

What's Wrong with These Methods ?

Conceptual Problem:

- They either do not guarantee that the number is 100 % prime
  - (it will be a prime with probability $1 - 2^{-80}$).
- Or they are not guaranteed to terminate in polynomial time.
- Or both…

Perfectly good for cryptographic applications.

Nothing wrong.

Part 2.2.

DH/ElGamal Key Generation

ElGamal and DH Key Generation in Z_p^*

Setup: g, a generator of Z_p^*.

- p prime > 1024 bits.
  - TWICE BIGGER THAN FOR RSA KEYS!
- How to find a generator ? NO EASY METHOD KNOWN (!).
  - Method 1: Use a prime with p=2p'+1 or similar.
  - Method 2: Requires to factor p-1.

Fact [by Lagrange Thm]: if for all prime $p_i \mid p-1$ we have $g^{(p-1)/pi} <> 1$ then g is a generator.