Algoritmi di Crittografia

Corso di Laurea Magistrale in Informatica

A.A. 2018/2019

Algoritmi di Crittografia

1. Randomness
   - Basic probability
   - Random (and Pseudo-Random) Number Generators

Algoritmi di Crittografia

1. Randomness
   - Basic probability
   - Random (and Pseudo-Random) Number Generators

Randomness and cryptography

- Modern crypto applications would be simply not possible without the availability of some source of *randomness*
- For instance, randomness is required to generate:
  - asymmetric keys in RSA-based protocols;
  - symmetric keys in SSH sessions;
  - stored salt data to be hashed with passwords;
  - initialization vectors to a block cipher's mode of operation.
- As we shall see, the ideal source of randomness for these (and other crypto) applications produces sequences of bits that are: (1) *independent*, (2) *uniformly distributed*, (3) always ready to hand
- Not as easy as it might sound...
- But let's start with some basic probability

Probability (informal)

- The probability of an event is a measure of “how likely is the occurrence of that event”
- Such measures are normalized values in the interval $[0, 1] \subseteq \mathbb{R}$
- Over discrete set of events, 0 means “impossible” while 1 means “certain”
- Over the reals (e.g., when dealing with time) things are little more complicated
- In computer science applications (and hence in Cryptography) the sets of events are always discrete
- But what exactly is the probability of an event? How can we assign such numbers to the events that might occur?
- Not always easy an answer

Probability (informal)

- Start by considering an experiment involving randomness (e.g., tossing a coin, rolling a dice, ...)
- Each possible outcome (e.g., the face which the coin ends up showing) is an *elementary event*
- Each elementary event is given (here is where we can make mistakes ...) a probability, i.e., a real number in $[0, 1]$
- The constraint is that the sum (finite or infinite) of the probabilities to all the events must be 1
- An *event* is then any subset of the set of all the elementary events
- Example, “the dice ends up showing a face with an even number of dots” is an event composed of 3 elementary events
- The probability of an event is the sum of the probabilities of the corresponding elementary events

Probability (decently formal)

- To reason about probability we need the following ingredients
- We are given a set Ω, finite or countably infinite, called the *sample space*
- We consider a function (a *measure*) over 2^Ω with values in $[0, 1]$; call them p (for probability...)
- p is defined for any $A \subseteq \Omega$ (i.e., $A \in 2^\Omega$)
- In particular, p is defined for any $\omega \subseteq \Omega$ such that $|\omega| = 1$ (ω is an elementary event)
- For any $A \subseteq \Omega$, it holds that

$$p(A) = \sum_{\omega \in A} p(\omega)$$

- If A is empty, $p(A) = 0$ while, if $A = \Omega$, $p(A) = 1$

Example

- We consider the experiment of tossing a coin and denote with *head* and *tail* the corresponing elementary events
- $\Omega = \{head, tail\}$
- Assuming the two outcomes are equally likely, and that it is impossible for the coin to land on the edge, we can set:

$$p(head) = p(tail) = \frac{1}{2}$$

Probability distributions

- The “form” of the probability function is known as the (probability) *distribution*
- When all the elementary events have the same probability (which implies that the sample space is finite) we speak of *uniform* distribution
- The distribution of the coin tossing example of the previous slide is uniform
- Uniform distributions are the most important in cryptographic applications, but many different (and important) distributions exist
- We will see one in the next two slides

Poisson distribution

- Suppose you (as a web server admin) are interested in the following question “How many http requests will arrive in the next t seconds”?
- Clearly, even if we may sensibly assume that the number of requests will be a finite quantity, we cannot place an à priori bound
- Hence, we want to assign a probability to all the events: no request (in t seconds), just one request, exactly two requests, and so on
- To do this, we need a model, which can be more or less accurate
- We assume to know the *rate* with which http requests arrive, a quantity that we denote by λ

Poisson distribution

- Let us define:

$$p(k \text{ requests in } t \text{ secs}) = \frac{(\lambda t)^k}{k!} \mathrm{e}^{-\lambda t}$$

- Since $p(k \text{ requests in } t \text{ secs}) \geq 0$ and

$$\sum_{k=0}^{\infty} p(k \text{ requests in } t \text{ secs}) = \sum_{k=0}^{\infty} \frac{(\lambda t)^k}{k!} \mathrm{e}^{-\lambda t} = 1$$

  it follows that p is a valid probability function
- Under reasonable assumptions, it is also a good choice, meaning that it produces accurate estimates of the process under analysis

Random variables

- Random variable represent a very useful way of dealing with sample spaces
- Consider the experiment of rolling a couple of dice
- What are the elementary events here?
- Clearly, a dice rolls and (when it ends rolling) the important fact is the number of dots in its up-facing side
- So, to describe one possible outcome, we should rigorously (but quite tediously) write something like:

  The first dice ended up rolling with the up-facing side showing 3 dots and the other dice ended up rolling with the up-facing side showing 2 dots

- Why not to say simply: one (dice) is 3 and the is 2?
- Because 3 and 2 are numbers, not events
- Here is where random variables come up

Random variables

- A random variable maps events to a numeric space S
- In S it is easier to make quantitative reasonings
- Random variables also make it possible to describe events in an easier way
- Lex X be the random variable that maps the 6 possible outcomes (elementary events) of the experiment of rolling a dice
- $X = 2$ is then a (very convenient) way to denote the elementary event: “The dice ended up rolling with the up-facing side showing 2 dots”
- Similarly, $x \leq 3$ denotes the event: “The dice ended up rolling with the up-facing side showing 1, 2, or 3 dots”

Random variables

- If X is a random variable and $x \in S$, then the probability that X has the value s, denoted $p[X = s]$, is

$$p[X = s] = \sum_{\omega \in \Omega | X(\omega) = s} p(\omega)$$

  i.e., is the sum of the probabilities of all the elementary events that maps to s
- Let X be a random variable that maps the outcomes of rolling two dice to $S = \{2, 3, \ldots, 12\}$ in the obvious way, namely:

$$\begin{array}{rcl} X(\text{⚀}, \text{⚀}) & = & 2 \\ X(\text{⚀}, \text{⚁}) & = & 3 \\ & \ldots & \\ X(\text{⚅}, \text{⚅}) & = & 12 \end{array}$$

Random variables

- But then for instance:

$$p[X = 4] = p(\text{⚀},\text{⚂}) + p(\text{⚁},\text{⚁}) + p(\text{⚂},\text{⚀})$$

- We can also define the value of X using a richer mathematical armory:

$$p[X \leq 3] = p(\text{⚀},\text{⚀}) + p(\text{⚀},\text{⚁}) + p(\text{⚁},\text{⚀})$$

or

$$\begin{aligned} p[X > 7 \text{and } X \leq 10] \quad = \quad & p(\text{⚁},\text{⚅}) + p(\text{⚂},\text{⚄}) + p(\text{⚂},\text{⚅}) + \\ & p(\text{⚃},\text{⚃}) + p(\text{⚃},\text{⚄}) + p(\text{⚃},\text{⚅}) + \\ & p(\text{⚄},\text{⚂}) + p(\text{⚄},\text{⚃}) + p(\text{⚄},\text{⚄}) + \\ & p(\text{⚅},\text{⚁}) + p(\text{⚅},\text{⚂}) + p(\text{⚅},\text{⚃}) \end{aligned}$$

Independence

- This is the last notion we recall explicitly, given its importance in cryptography
- Consider two events, E_1 and E_2, not necessarily defined over the same sample space
- We say that E_1 and E_2 are independent events if the occurrence of E_1 does not affect the probability of E_2 occurring, and vice versa
- If we toss a coin and roll a dice, the probability that the coin lands heads is $\frac{1}{2}$ (assuming the coin is not biased), independently of the outcome of the dice experiment

Independent random variables

- Consider rolling two six-face dice and assume that the dice are not biased, so that we can fairly assume that all the possible outcomes have probability $\frac{1}{6}$.
- Let X and Y be random variables that refers to the first and second dice experiment, respectively
- Then, X and Y being independent amounts to the following property:

$$p(X = x \text{ and } Y = y) = p(X = x) \cdot p(Y = y) = \frac{1}{6} \cdot \frac{1}{6} = \frac{1}{36}$$

  where clearly $x, y \in \{1, 2, 3, 4, 5, 6\}$
- If X and Y are independent we may also write, e.g.:

$$p(X \leq 4 \text{ and } Y > 2) \quad = \quad p(X \leq 4) \cdot p(Y > 2) = \frac{4}{6} \cdot \frac{3}{6} = \frac{1}{3}$$

Algoritmi di Crittografia

1. Randomness
   - Basic probability
   - Random (and Pseudo-Random) Number Generators

A first simple example

- For cryptographic applications, it is crucial to have sources of randomness, i.e., logical or physical devices that can produce (measurable) random events in some space
- What ultimately matters is to map the available randomness to sequences of *independently generated bits* (or, simply, *independent bits*)
- Suppose we need a randomly chosen binary sequence of length n (say, to generate a session key as part of some symmetric protocol)
- All the possible 2^n sequences must have the same chance to be chosen
- Assume also, for simplicity, that $n = 4$

A first simple example

- Suppose to this end we use a Random Bit Generator, i.e., a piece of software that, each time is called, returns 1 with probability p and 0 with probability $1-p$ independently of *past history*
- In other words, the bits are independently generated
- Our question is: what is the probability of generating (say) 1011 as key?
- Well, this is easy: since the bits are independent, the probability is $p(1-p)pp = p^3(1-p)$
- In general, a key with k ones and $4-k$ zeroes will show up with probability:

$$\binom{4}{k} p^k (1-p)^{4-k}$$

A first simple example

- Now we ask a different question, with a clear cryptographyc flavor: how hard would be for $\mathcal{E}$ to guess our session key?
- Suppose $p = 1$ and (clearly) $1 - p = 0$. In this case only one key is possible, namely 1111 which requires no guess at all...
- Suppose now that $p = 0.9$. All the 16 4-bit long sequences have their chance now. But the sequence 1111 is still the best guesss for $\mathcal{E}$, who will be right approximately 66% of the times
- So, what is the value of p that cause $\mathcal{E}$ having a hard time to guess correctly?

Surprise and ... information

- Let's analyze things from another perspective
- Assume $\mathcal{E}$ knows the value of p and consider again the state of affairs where $p = 1$.
- In this case releasing to $\mathcal{E}$ the "information" that the key is 1111 actually amounts to giving her no-info at all
- Suppose now that $p = 0.9$. This time releasing the same info as above would indeed carry some information, however one that definitely would not knock Eve out
- It then seems that the question raised in the previous slide can be rephrased as follows: what is the value of p that implies maximum gain of information from the outcome?

Entropy

- Entropy is a well-known concept in thermodynamics, which is however not relevant to us
- In *Information theory*, the entropy of a random event is a measure of the average information that one acquires when the outcome is revealed
- ... and this is exactly what we need
- If the event has k possible outcomes, and $p_1, p_2, \ldots, p_k$ denote their probabilities, then the entropy is defined as

$$\sum_{i=1}^{k} p_i \log_2 \frac{1}{p_i} = -\sum_{i=1}^{k} p_i \log_2 p_i$$

- The values we are looking for are then those p_i that maximize the above quantity, under the constraints: $p_i \geq 0$ and $\sum_{i=1}^{k} p_i = 1$

Entropy

- If $k = 2$, hence $p_1 = p$ and $p_2 = 1 - p$, it is an easy excercise to prove that entropy is maximum for $p = \frac{1}{2}$
- The result carries over the general case, namely $p_i = \frac{1}{k}$, for $i = 1, \ldots, k$, are the values that implies maximum entropy
- For our toy example, it $p = \frac{1}{2}$, then all the keys have the same probability of $\frac{1}{16}$ and the entropy is

$$16\left(-\frac{1}{16}\log_2\frac{1}{16}\right) = -\log_2\frac{1}{16} = \log_2 16 = 4$$

- In other words, in case of $p = \frac{1}{2}$ we learn four bits of information and this is supposed to be maximum information
- Indeed, it makes sense: how would it be possible to get more than four bits of information when the key is exactly four bit long?

Entropy

- Any other (i.e., non-uniform) distribution gives lower entropy
- The same happens if the bits are not independent
- For instance, suppose the probability of the next bit being 1 is $1/4$; then the entropy E associated to the next two bits is:

$$\begin{aligned}
E &= -\mathrm{prob}(00)\log\frac{1}{\mathrm{prob}(00)} - \mathrm{prob}(01)\log\frac{1}{\mathrm{prob}(01)} - \\
&\qquad \mathrm{prob}(10)\log\frac{1}{\mathrm{prob}(10)} - \mathrm{prob}(11)\log\frac{1}{\mathrm{prob}(11)} \\
&= \left(\frac{3}{4}\right)^2 \log\left(\frac{4}{3}\right)^2 + 2\frac{3}{4^2}\log\frac{4^2}{3} + \left(\frac{1}{4}\right)^2 \log 4^2 \\
&\approx 1.62
\end{aligned}$$

Random Number Generators (RNG)

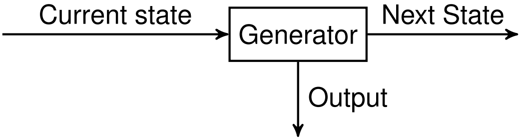
- The ideal source of randomness would then be a Random Bit Generator which always returns 0 or 1 with uniform probability (i.e., independently of past bits)
- In general, in the crypto literature we speak of *Random Number Generators*, rather than of random bit generators, irrespectively of the actual device being oriented to bits or numbers
- Aside for terminology, to generate truly random bits is all but easy in practice
- For non-crypto applications that embody a probabilistic model of some sort, *pseudo-randomness* is almost always adequate, and pseudo-randomness is much easier to obtain
- For crypto application pseudo-randomness is almost unavoidably the origin of disasters

The problem in a nutshell

- Having random bits ready at hand (and at will) when they are needed is problematic
- True randomness can be extracted by the physical environment (including quantum mechanical phenomena)
- There might be availability problems (we need physical devices attached to the computer)
- Also, it might not be easy to transform the environmental randomness in a perfectly uniform and independent sequence of bits
- Gathering randomness from the computer itself (e.g., from disk activity, keyboard or mouse inputs) is clearly easier but might not be completely secure
- In all cases, the rate at which the random bits can be produced is not sufficient in many application settings (e.g., think of Amazon web servers...)

A general solution

- Combine true RNGs with good Pseudo-Random Number Generators (*PRNGs*)
- Essentially, the RNG is used to “seed” a cryptographic PRNG (and later to collect other randomness, if and when available)
- A cryptographic hash function or a cipher is then used to return pseudo-random data (the output) and to update the internal state



Simple PRNG operations

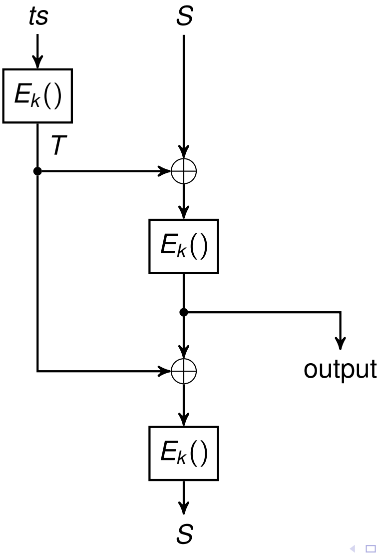
Outline

- This “architecture” prevents an attacker from computing the internal state/seed from the output
- This is usually sufficient to prove *computational security* (rather than *unconditional security*)
- We shall describe three different (allegedly...) *Cryptographically Secure PRNGs* (*CSPRNGs*)
    1. The *ANSI X9.17* standard, a PRNG which was widely adopted especially in banking applications
    2. The *Fortuna* PRNG, used in Windows® systems
    3. The `/dev/random` and `/dev/urandom` generators under Linux and other *nix systems

The ANSI X9.17 generator

- Requirements:
  - A secret key k generated at initialization time
  - A *Triple-DES* symmetric block-cipher
  - A short initial random seed S (64 bits), which is (part of) the internal state
- When pseudorandom bits are required:
  1. Compute a timestamp ts of current date and time T at maximum possible resolution (e.g., microseconds)
  2. Using the encryption function of 3DES with key k:
     - compute $T = E_k(ts)$
     - compute the output value $O = E_k(T \oplus S)$
     - update the internal state: $S = E_k(T \oplus O)$
  3. Return O

ANSI X9.17 block diagram



A Python code for ANSI X9.17

```
from datetime import datetime
from Crypto.Cipher import DES3
from Crypto import Random
from Crypto.Util.strxor import strxor

class ANSIX917:
    def __init__(self, keylen):
        assert(keylen == 24 or keylen == 16)
        key = Random.new().read(keylen)
        self.__state = Random.new().read(keylen)
        self.cipher = DES3.new(key, DES3.MODE_ECB)
```

A Python code ANSI X9.17 (cont.d)

```
class ANSIX917:
...
def __iter__(self):
  return self

def __next__(self):
  fmt = '%y-%m-%d %H:%M:%S.%f'
  ts = datetime.now().strftime(fmt)
  T = self.cipher.encrypt(ts)
  out = self.cipher.encrypt(strxor(T, self.__state))
  self.__state = self.cipher.encrypt(strxor(T, out))
  return out
```

Usage

```
In  [1]: from ANSIX917 import ANSIX917

In  [2]: PRNG = ANSIX917(keylen=16)

In  [3]: next(PRNG)
Out[3]: b'\x8c\x98\xbf~\xf3\xae\xabrB\x87\xd8\xe7\x

In  [4]: next(PRNG)
Out[4]: b'l0!\xaa:!\x8b\xe2d\xf34\xa3a\x19\xc0*|\x1
```

Security issues

- At the macroscopic level the goals are obvious: it should be impossible for an attacker to recover both past and future generated bits
- Being able to to protect the past is known as *backtracking resistance*
- Success in protecting future bits is the property of *prediction resistance*
- For backtracking resistance the general idea is to use *one-way* transformations: either a cryptographic hash function, which is one-way, or a cipher, which is not invertible without the key
- For prediction resistance, a good strategy is to regularly update the internal state with fresh random bits, a solution that the ANSI X9.17 PRNG does not implement

Attacks to PRNGs

- We have a *direct cryptoanalytic attacks* when the attacker is able to distinguish the outputs from truly random bists
- If the attacker, at some time, has access to the internal state of the PRNG (for whatever reason), then s/he will try to exploit this knowledge for as long as possible with *state compromise extension attacks*
- Finally, cryptoanalytic attacks can take advantage of the attacker being in some control of (possible) PRNG inputs; in this case we speak of *input-based attacks*
- We now present a simple case study with the hope to shed light upon some of the above points

A known attack to ANSI X9.17 PRNG

- Tecnically, it's a state compromise extension attacks
- However, it also exploits the low entropy in the PRNG inputs (and hence it can also be considered an input-based attack)
- This attack is directed towards discovering the ANSI X9.17 state using the *meet-in-the-middle* attack tecnique
- It works under the assumption that the attacker has somehow learned the key K (due to a flaw in the implementation, a leakage of information, ...)
- The attacker needs also two consecutive outputs from the PRNG

A known attack to ANSI X9.17 PRNG

- The attack works as follows.
- We know that

$$\begin{aligned} T_i &= \mathbf{E}(K, t) \\ O_i &= \mathbf{E}(K, T_i \oplus S_i) \\ S_{i+1} &= \mathbf{E}(K, T_i \oplus O_i) \end{aligned}$$

where t is current *timestamp* while O_i is the i-th output.

- It follows that S_{i+1} can be computed in two different ways, namely:

$$S_{i+1} = \mathbf{E}(K, T_i \oplus O_i) \tag{1}$$

and

$$S_{i+1} = \mathbf{D}(K, O_{i+1}) \oplus T_{i+1} \tag{2}$$

A known attack to ANSI X9.17 PRNG

- The point is that we (the alleged attackers...) do not know at which time the outputs and the new states were computed, i.e., which timestamps t were used
- This is, however, a low entropy input to the PRNG
- If we can approximate t, say within a second or few seconds (which is highly reasonable), then we are left with about $\log_2 10^6 \approx 20$ bits of entropy, since we assumed resolution of microseconds
- This is well affordable for any commodity PC so that we can mount the following search:
  1. compute (1) and (2) for different values of t
  2. stop when (and if), for some t_1 and t_2:

$$\mathbf{E}(K, t_1 \oplus O_{i+1}) = \mathbf{D}(K, O_{i+1}) \oplus t_2$$

A Python code for the attack (1)

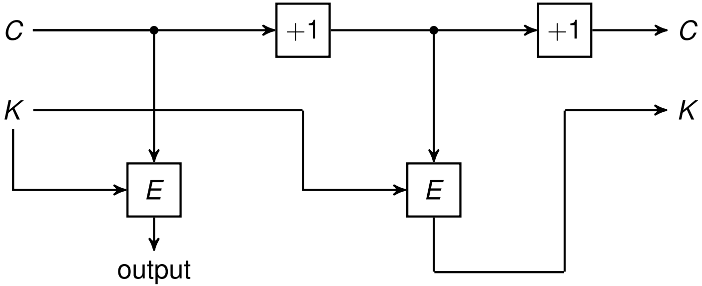
```
def timestamps(T):
    def advance():
        from datetime import timedelta
        nonlocal T
        T += timedelta(microseconds=1)
        return T.strftime(fmt)
    for i in range(1000000): yield advance()

def PSCAttack(ANSIX917gen, key):
  fmt = '%y-%m-%d %H:%M:%S.%f'
  cipher = DES3.new(key, DES3.MODE_ECB)
  t = datetime.now()
  o1, o2 = next(ANSIX917gen), next(ANSIX917gen)
  L = []               # A list of candidate seeds
  ts = timestamps(t)   # A generator of timestamps
```

A Python code for the attack (2)

```
from Crypto.Util.strxor import strxor
from bisect import bisect_left
for j in range(100000): # Should be sufficient
    t = next(ts)
    T = cipher.encrypt(t)
    s1 = cipher.encrypt(strxor(T,o1))
    L.insert(bisect_left(L,s1),s1)
    d = cipher.decrypt(o2)
    s2 = strxor(d,T)
    idx = bisect_left(L, s2)
    if idx!=len(L) and L[idx]==s2:
        print("Success'')
␣␣␣␣␣␣␣␣␣␣break
```

A more sophisticated design: The Fortuna CS-PRNG

- So named after the Roman goddess of chance
- Fortuna's internal state is a pair (K, C), where K is a 256-bit key while C is a 128-bit counter
- The generator works as depicted below:



Output and key update in Fortuna. E is the AES (or an AES-like) encryption function.

Fortuna's detailed operations

Initialization Assign initial values to the counter C and the key K

Reseeding Update the internal state (both C and K)

Block generation Using the AES block cipher, generate a number of 128-bit (16 bytes) blocks sufficient to satisfy the user's request

Random data generation Generate the number of random data requested by the user

Collection Collect real random data from entropy sources, used to reseed the generator

Initialization and reseeding

- Initialization is exceedingly simple

```
K = b'0' # i.e. bytes
C = 0
return (K,C) # State of generator
```

- Reseeding mixes the current key with (fresh random) data S
- Fortuna uses double SHA256 hashes (which simply means SHA256 twice)

```
K = SHAd256(K||S) # || here is string/bytes concat
C = C + 1
```

Block generation

- The input is the number k of required blocks
- State (C, K) is clearly available (e.g., global or instance variables/attributes)
- E is the AES encryption function

```
def GenerateBlocks(k)
    out = b''
    for i in range(k):
        b = E(K,C)
        out = out || b
        C += 1
    return output
```

- Block generation is meant as an internal function

Random data generation

- The input is the number *n* of random bytes required
- *n* must not exceed 2^{20} (i.e., at most 1MB of data can be generated per-request)
- The reason is that increasing the size would also increase the statistical divergence from true randomness

```
if n>2**20:
    raise some error
B = GenerateBlock(ceil(n/16))[:n]
K = GenerateBlock(2) # Update the key
return B
```

Collection

- This is the most important (and difficult) aspect of a CSPRNG
- Collecting real randomness from the environment requires (if not dedicated hardware, at least) special enablement of various drivers
- From the perspective of the CSPRNG, randomness collection thus means *storing* and *making available* the entropy collected from external sources (e.g., the drivers)
- Typically, entropy is gathered from events like mouse movements and keystrokes as well as other various timing sources
- In Fortuna, each source distributes the entropy it collects (evenly) over 32 different pools
- In this way, unless the attacker has full control of all the sources, s/he cannot reconstruct the contents of each pool

Reseeding reviewed

- Reseeding clearly uses data coming from the pools
- Reseeding is done when pool P_0 has enough data
- Reseeding number k uses data from all the pools P_j such that 2^j divides k
- Thus, for instance, pool P_0 is used at each reseeding while P_2 is used at reseedings 4, 8, 12 and so on
- This results in a sort of frequency/amount of entropy tradeoff for the attacker who snatched the internal state
- Suppose s/he does not control source 0: then the system is reseeded too frequently with data s/he does not know
- Suppose now the attacker does not control source 1 (or 2); then the generator is reseeded less frequently with data s/he does not know, but in much larger quantity

PyCrypto implementation of Fortuna

```
from Crypto.Random.Fortuna import FortunaGenerator
from Crypto.Util.py3compat import b
from binascii import b2a_hex

fg = FortunaGenerator.AESGenerator()
fg.reseed(b(seedstring))
# Initial key is computed as
# SHAd256(b'\x00\x00...\x00'+b(seedstring))
print(fg.key)
fg.counter.next_value() # --> 1
b2a_hex(fg.pseudo_random_data(16))
print(fg.key)
fg.counter.next_value() # --> 4
```

Generating random bytes in Linux

- Under Linux the user may extract high-quality cryptographic random bytes from the special files `/dev/random` and `/dev/urandom`
- Both files (actually, a character special files) harvest entropy from the same sources
- The fundamental difference is that requests to one of these (i.e., `/dev/random`) are *blocking*, while requests to the other are always immediately honored
- This means that if the generator has not sufficient entropy to satisfy the request, then the process that issued it is put oh hold
- Under some circumstances, this might be preferable

Using /dev/random and /dev/urandom from command shell

- The following command reads 24 bytes of (pseudo)random data from */dev/urandom* and prints it to the terminal using `base64` encoding (for readability)

  ```
  dd if=/dev/urandom bs=8 count=3 2>/dev/null|base64
  ```

  (redirection of `stderr` is required to keep the output clean)
- When using `/dev/random` it may be useful to keep an eye on the available entropy beforehand

  ```
  cat /proc/sys/kernel/random/entropy_avail
  ```

- ... and then using `/dev/random` under timeout guard

  ```
  timeout 5s dd if=/dev/random/ bs=....
  ```

  to avoid blocking (while possibly getting less data)

PRNGs in Python

```
from os import urandom
from Crypto.Random.OSRNG import os
from binascii import b2a_hex

urandom is os.urandom       # eval to True
r = b2a_hex(urandom(32))    # returns 32 bytes
                            # of random data
```