Cryptographically Secure Random Numbers

Uses of Random Numbers

- Encryption keys
- Password salts
- TCP/IP initial sequence numers
- TLS nonces

Cryptographically Secure Random Numbers

- It is important that cryptographically secure random number generators are used for these operations (symmetric key)
- (In other words, the random numbers can not be guessed).
- In some other uses this is not so important.
- For example a sequence of random numbers might be used to randomlly allocate machine learning data to a training set and a test set.
- In this case the requirements for cryptographically secure randomness would not be so great.

Random Number Generators

- Two types
- Truly random (TRNG)
  - Normally obtained from some physical process that is only observable to whoever generates the random number.
- Pseudo random (PRNG)
  - Algorithm based
  - For a given seed the sequence of numbers generated is determined

Random Number Generators

- True random numbers are always cryptographically secure but it is often difficult to supply enough truely random numbers.
- Hence there is a need to use pseudo random number generators as they are algorithm based and can generate large numbers of random numbers easily.
- It is important that these pseudo random number generators are cryptographically secure.
-

Truly Random Number Generator

- Utilizes physical processes.
- Linux maintains a pool of true random numbers generated by
  - the speed of key presses while typing
  - the least significant bit of digit voltage measurements
  - other non-deterministic and measurable processes.

Pseudo Random Number Generators

- Generated by an algorithm.
- Are they truly random?
- No.
- Input is a seed.
- Depending on this value, a sequence of (pseudo) random numbers are generated.
- The same sequence is generated if the seed is reused.

Pseudo Random Number Generators

- In software development, when a random number generator is used, a setSeed() method or function is often used for testing purposes.
- Ensures the same sequence of random numbers are generated every time the program is run.
- And the same output should be generated each time.
- Proper use would require that a truly random value is used to set the seed.

Cryptographically Secure Pseudo Random Number Generators

Cryptographically Secure Pseudo-Random Number Generator

- CSPRNG
  - Statistically Random - It appears random.
  - Unpredictable - Even with partial knowledge about the state or partial knowlede about the state of the generator, it should be computationally infeasible to predict past or future outputs.
  - Secure for Cryptographic use (resistent to attacks)

CSPRNGs

- Should start with a truly random seed.
- They use an algorithm that is deterministic given the seed.
- Are resilient against attacks using partial knowledge.
- Of course if the initial seed is known, then all subsequent values can be generated.
- Most PRNGs are reseeded with truely random numbers, at certain intervals.

Partial Knowledge Attacks

- To be cryptographically secure a PRNG must
  - Prediction resistence (pass the “Next-bit Test”)
  - Backtracking resistent (withstand a “State Compromise Extension” attack.)

Prediction Resistence - Next bit Test

- Given the first k bits of a random sequence there is no polynomial-time algorithm that offers a greater than 50% probability of correctly predicting the k+1 th bit of the sequence.

Backtracking Resistence - State Compromise Extension

- If the state of the PRNG at a certain time is revealed, it should not be possible to determine the numbers previously generated

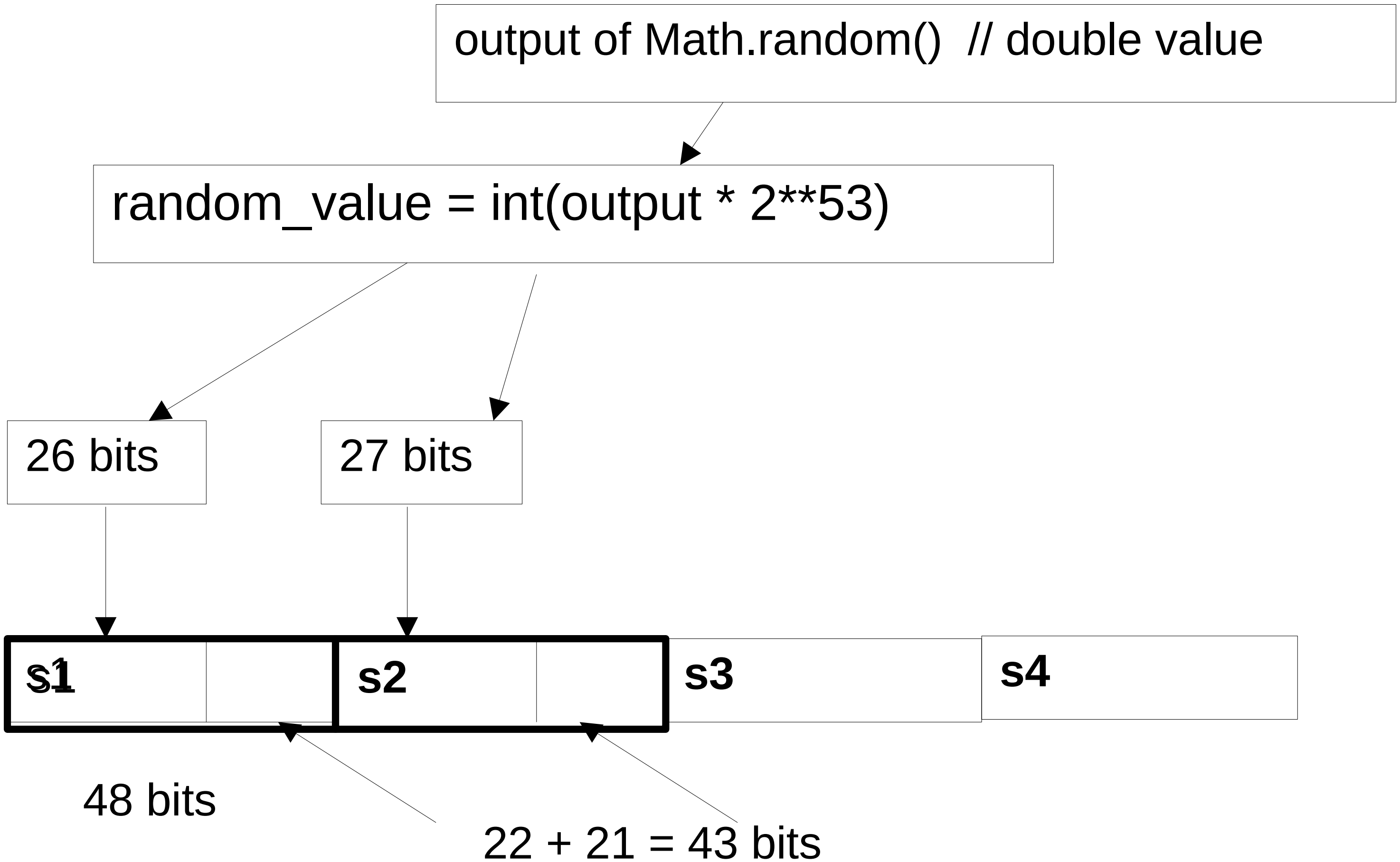
Case Study 1 – java.util.Random

- Math.random in Java.
- Given one number generated by the algorithm, it is relatively easy to calculate the next number generated. It is not cryptographically secure.
- $s_{n+1} = (a \cdot s_n + c) \bmod m$
- Parameters are fixed.
  - $m=2^{48}$
  - a=25214903917
  - c=11

Case Study 1 – java.util.Random

- The internal state of the algorithm (s_n) is a 48 bit value.
- An output from the algorithm is a 53 bit value and is obtained from the upper 26 and 27 bits of two successive (48 bit internal) states.
- So from a single output we know 26 bits of one internal state and 27 bits of the next state.

Case Study 1 – java.util.Random



Case Study 1 – java.util.Random

- We have 26 bits of one 48 bit value and 27 bit of the next 48 bit value.
- The values are related by
  - $s_2 = (a \,.\, s_1 + c) \bmod m$
- where a, c and m are known.
- Search space of size 2^{43} (22 bits for s1 and 21 bits for s2).
- This is straight forward for modern machines.
- Search until s_1 and s_2 are found that satisfy the reurrence relation.

Case Study 1 – java.util.Random

- Obviously java.util.Random does not satisfy the Next bit text.
- It is not cryptographically secure.

Case Study 2

- A hack of the hacker news website - https://news.ycombinator.com/
- Carried out with the permission of the website owner.
- When users logged on, they were give a unique id for the session.
- If the id is guessed, the logged on user can be impersonated.
- See
  - https://blog.cloudflare.com/why-randomness-matters/
  - https://news.ycombinator.com/item?id=639976

Generating Session IDs

- A cryptographically secure PRNG was used to generate the sequence of session IDs handed out to the user.
- The problem was with the seed used.
- Which was the time in milliseconds when the software was last started.
- What is the problem with this?

Obtaining the PRNG seed

- The hacker was able to crash the system and obtain the time it restarted to within a minute,
- That was 60,000 seeds to be checked.
- Attacker logged in and obtained a sessionID.
- Checked 60,000 seeds to see which of them generated that sessionID.
- Used the seed and knowledge of what PRNG algorithm was used to reproduce the sequence of sessionIDs.

Recap - PRNG weaknesses

- Algorithm is weak (java.util.Random)
- Seed used is bad (case study 2)

Two older PRNGs

Two older PRNGs

- Linear Congruential Generators (LGC)
- Linear Feedback Shift Register (hardware)

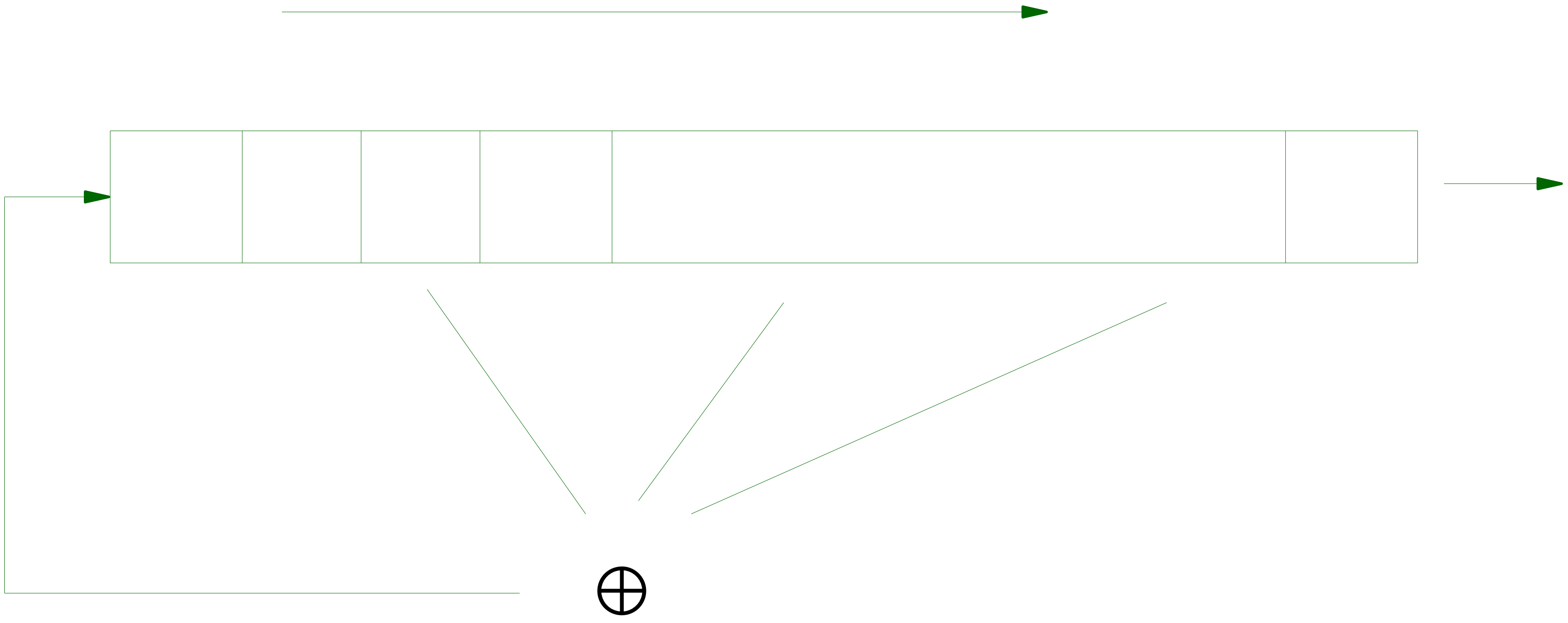
Linear Congruential Generators (LCG)

- (Now avoided)
- $x_n = (a * x_{n-1} + c) \bmod m$
- Three parameters
  - a (multiplier)
  - c (increment)
  - m (modulus)
- x_0 is the seed

LCG

- Will pass tests for randomness
- Repeats after some period
- With appropriate parameters the period is known and long
- Quality of the output is sensitive to the choice of parameters.
- Poor choices of parameters have led to bad PRNGs.
- Its main problem is that it is very easy to choose bad parameters.

Linear Feedback Shift Register.



Linear Feedback Shift Register.

"A LFSR is most often a shift register whose input bit is driven by the XOR of some bits of the overall shift register value."

- The bit positions that affect the next state are called the taps.

Content Scrambling Systems

- Based on a LFSR.
- For encrypting DVDs.
- Badly broken.
- Hardware stream cipher (designed to be easily implemented in h/w.
- Uses a linear feedback shift register (hardware) to generate random numbers.

LFSR based Stream Ciphers

- DVD encryption (CSS): 2 LFSRs
- GSM encryption (A5/1,2): 3 LFSRs
- Bluetooth (E0): 4 LFSRs
- (All broken)
- All implemented in hardware!!

Sources of Random Numbers in Linux

Sources of Random Numbers in Linux

- Entrophy - A measure of the disorder or randomness in a closed system.
- */dev/*random is a source of true entropy.
- True entropy (truely random numbers) originates from device drivers, keyboard, mouse, variations in fan noise, disk noise etc.
- Needed to seed pseudo random number generators.

TRNG - /dev/random

- Source of true entropy.
- Gives truly random numbers read from physical phenomena (mouse, keyboard, disks, voltages etc.)
- The amount of true random data is often insufficient.
- reading */dev/*random will block if there is insufficient truely random data.

PRNG - /dev/urandom

- Unlimited or non blocking.
- Uses a CSPRNG (crytographically secure PRNG) seeded periodically using the pool of entropy (from /dev/random).
- "/dev/urandom" is fine to use in most cases when a cryptographically secure PRNG is required.

Random Numbers in Java

java.util.Random

- Math.random()
  - Calls java.util.Random
- java.util.Random
  - Uses the system clock as the seed (Attacker could determine if they know the time the random number was generated.)
  - Generates 48 bit values
  - Uses a Linear Congruential Generator

java.util.Random

- It is possible to generate future random numbers from a single random number generated by java.util.Random.
- Should not be used when security is a concern.

java.security.SecureRandom

- Can be internally implemented with a PRNG with a TRNG seed.
- Or as a TRNG.
- Or a combination of both.
- extends java.util.Random

java.security.SecureRandom

- A cryptographcally strong random number generator.
- Generates 128 bit values (2^128 values can not be generated in a reasonable amout of time)
- Seed is taken from the OS source of true random numbers (e.g. */dev/random)*
- Should never be explicitly seeded.

java.security.SecureRandom

```
SecureRandom random = new SecureRandom();
System.out.println(random.nextInt(1000));


// or to generate a seed for another PRNG
byte seed[] = random.generateSeed(20);
```

SecureRandom Algorithms

```java
final Set<String> algorithms =
            Security.getAlgorithms("SecureRandom");

for (String algorithm : algorithms) {
        System.out.println(algorithm);
}

final String defaultAlgorithm =
                    new SecureRandom().getAlgorithm();

System.out.println("default: " + defaultAlgorithm)
```

Implementations of SecureRandom on Linux

```
DRBG
SHA1PRNG
NATIVEPRNGBLOCKING
NATIVEPRNGNONBLOCKING
NATIVEPRNG
default: NativePRNG
```

- Always best to used the default implementations unless you really know what you are doing.

Native PRNG

- `Native PRNG is by default NATIVEPRNGNONBLOCKING (/dev/urandom)`
-
- Native PRNG:-
  - Source – /*dev/u*random
  - Seed source - /*dev/*random

NativeNonBlockingPRNG

- Algorithm used depends on the version of Linux/Unix.
- Later version of Linux uses an algorithm based on the stream cipher ChaCha20.

Other Providers

- SHA1PRNG - Java implementation.
- Deterministic Random Bit Generator (DRBG)

SHA1PRNG

- Gets a true random number using an entropy source.
- Concatenates it with a 64-bit counter which increments by 1 on each operation.
- Then gets the SHA1 hash value (160 bits)
- [A one way hash has the property that if just one bit is changed in the input then the output changes substantially.]
- DRBG is considered better.

DRBG

- Hash-DRBG: Uses cryptographic hash functions (e.g., SHA-256).
- HMAC-DRBG: Uses HMAC (Hash-based Message Authentication Code).
- CTR-DRBG: Uses Block Cypher
-

Review

Review

- True random numbers are obtained from physical sources. (TRNG)
- There is normally a limit on the amount of true random numbers available (true entropy)
- Hence pseudo-random number generators (PRNG) aneed to be used.
- Pseudo-random numbers are generated using a seed and an algorithm.
- A lot of PRNGs are not suitable for use in applications with security concerns, for example java.util.Random.

Review (cont)

- For applications that have security concerns, a cryptographically secure random number generator (CSPRNG) should be used.
- A CSPRNG is seeded from a source of true entropy (TRNG).
- A PRNG is a CSPRNG if :-
  - The algorithm is resistent to partial state attacks.
  - The algorithm is seeded from a source of true randomness.