Topics for today: Pseudo-Random Number Generators (PRNGs), Extractors

**Pseudo-Random Number Generators**

(PRNG/PNRG/CS-PRNG)

\[ y = \text{PRG}(x) \text{ where } y \text{ is the (arbitrarily long) output} \]
\[ \text{x is a d-bit long input/seed} \]

A PRG maps a fixed input to an arbitrarily long output: \(\{0, 1\}^d \rightarrow \{0, 1\}^*\)
PRGs are deterministic - given the same \(x\), they will output the same \(y\)

Security Defn: A PRG is secure (CS-PRG) if it is infeasible to distinguish the output \(y\) from a truly random bitstring of the same length, assuming that the seed is unknown.

In order to meet this definition today, the input size \(|x| > 112\) bits (NIST standard)
To see why, imagine the following PRG with \(|x| = 2\):
\[
\begin{align*}
\text{PRG}(x) &= \begin{cases} 
\text{PRG}(1) = 01110101 \\
\text{PRG}(2) = 11000010 
\end{cases}
\end{align*}
\]
We can distinguish this PRG from a random process because we will only ever see these two inputs.

The above security definition implies two other properties:
1. It is infeasible to predict the next bit if you don’t know \(x\).
   Alice: \(\text{PRG}(\text{seed}) \xrightarrow{b_1} | Bob = b_1 b_2 \ldots b_n \xrightarrow{b_2} | \rightarrow \text{infeasible for Bob to predict } b_3\)
2. It is infeasible to recover the seed given the output.

Examples of PRGs:

- Linear Feedback Shift Register
- Cellular Automata
- AS/1 AS/2 (insecure)
- RC4 (secure)
- “Counter Mode”

\{ Secure \}
\{ Insecure \}

For security to hold, the seed must be unpredictable.

- This is often a problem in practice
- Early 90s: netscape broke SSL (used predictable OS values)
- Late 2000s: Debian breaks SSL/SSH (new patch reduced seeds entropy)
- last year constrained devices/embedded systems break SSL
**Extractors**

Primitive used to help seed PRGs

\[ y = \text{Ext}_i(x) \] where \( y \) is the (fixed length) output

- \( \text{Ext} \) is a function
- \( x \) is variable length input
- \( i \) is a nonsecret key (that we don’t care about)

An Extractor maps a variable length input to a fixed length output: \( \{0, 1\}^* \rightarrow \{0, 1\}^d \)

If \( x \) has min-entropy greater than \( d \) (typically \( H_0 > 2d \)), then it is infeasible to distinguish \( y \) from a \( d \)-bit truly random string.

How they are used with PRGs:

\[ y = \text{PRG}(\text{Ext}(x)) \]

Used when we know that data has some randomness, but we’re not sure how exactly to pull it out.

Unix: /dev/random  Windows: CryptGenRandom

Process: at the end we want something really large and random, but we have some really large and semi-random

\[ \rightarrow \] so we collapse it down (Ext) then build it back up (PRG)

Proof of concept Extractor:

Biased Coin

\[
\begin{align*}
\text{Heads } p_h &= 0.6 \\
\text{Tails } p_t &= 0.4
\end{align*}
\]

more predictable than a truly random coin, so less random

Entropy is a measure of “unpredictableness”

\[ \rightarrow \] it is in units of bits

\[ \rightarrow \] Entropy of \( m \) bits has the same “randomness” as flipping a coin \( m \) times

Average Entropy (Shannon Entropy):

\[
H(p_1, p_2, \ldots, p_n) \leftarrow \text{Set of probabilities of all events that can occur} \nonumber \\
= - \Sigma p_i \log_2(p_i)
\]

For our biased coin:

\[
= - [0.6 \log_2(0.6) + 0.4 \log_2(0.4)]
\]

\[ = 0.736 \]

Entropy of 20 biased coin flips:

\[ = 20 \times 0.736 \]

\[ = 14.6 \]

Von-Neumann Extractor:

\[ \rightarrow \] not ideal, gives a lower entropy

Note: \( \perp \) means there is no output

Given \( b_1 b_2 \), output \( y_k = \begin{cases} 
\text{HH} & \downarrow \text{pr} = 0.6 \cdot 0.6 \\
\text{HT} & \text{H} \quad \text{pr} = 0.6 \cdot 0.4 \\
\text{TH} & \text{T} \quad \text{pr} = 0.4 \cdot 0.6 \\
\text{TT} & \downarrow \text{pr} = 0.4 \cdot 0.4 
\end{cases} \]

Essentially we extract events which happen with the same probability (in this case the sequence going from \( H \) to \( T \) or from \( T \) to \( H \) both have probability 0.6*0.4 = 0.24)
Von-Neumann Problems:
→ exploits knowledge of x (or its distribution). i.e knows that it’s a biased coin
→ has a non-deterministic output size

Instead we use:
→ Any x (where $H_0(x) > 2d$)
→ always returns d-bits

$H_0$ is the min-entropy. This is the entropy in the worst case.
$H_0 = -\Sigma 1^1 \log_2(p_{\text{max}})$ because we only consider one event, its probability is 1
$= -\log_2(p_{\text{max}})$

Summary

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash</td>
<td>${0, 1}^*$</td>
<td>${0, 1}^d$ PR, CR, (RO)</td>
</tr>
<tr>
<td>PRG</td>
<td>${0, 1}^d$</td>
<td>${0, 1}^*$ Pseudo-Random Output</td>
</tr>
<tr>
<td>Ext</td>
<td>${0, 1}^*$</td>
<td>${0, 1}^d$ Pseudo-Random Output</td>
</tr>
</tbody>
</table>