SE 4472 / ECE 9064

Information Security

Week 12:

Random Number Generators

and

Picking Appropriate Key Lengths

Fall 2015
Prof. Aleksander Essex

Western Engineering

Random Number Generation

Where do keys come from?

- So far we've abstracted the idea of key generation. At times I've said something like "Alice needs a b-bit key so she flips b coins"
- So where to keys really come from?

Entropy

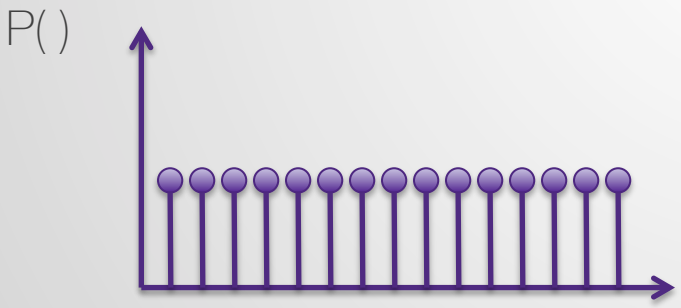
- Notion borrowed from information theory (which is borrowed from thermodynamics)
- For our purposes: it's random bits collected by your app/OS from various hardware sources like mouse position, time, network data, etc
- In UNIX systems its made available through /dev/random

Entropy

- Many of these sources individually are weakly random
- For example, your mouse is more likely to be near the center of the screen than the edge, and it's position remains static for long periods
- The sources are combined and sent through a randomness extraction process to provide large amounts of unbiased (uniform) output

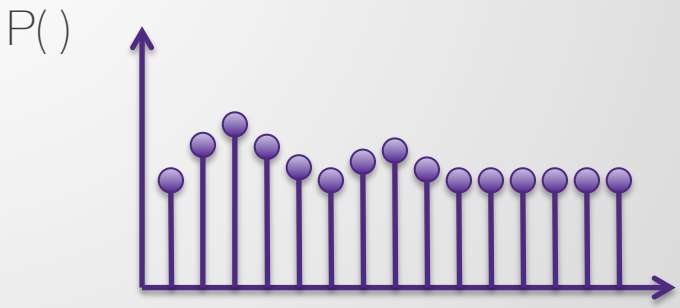
Uniform Random Numbers

- We need uniform random numbers to do cryptography. That is, each number in the range of possible numbers has an equally likely chance of being chosen



Uniform distribution

All numbers have equal probability of being chosen



Non-uniform distribution

Some numbers are more likely than others

Uniform Random Numbers

Reason 1: Brute force is maximally hard. If there was a bias in how keys were chosen, the attacker could search a smaller set of numbers with a greater chance of success

Uniform Random Numbers

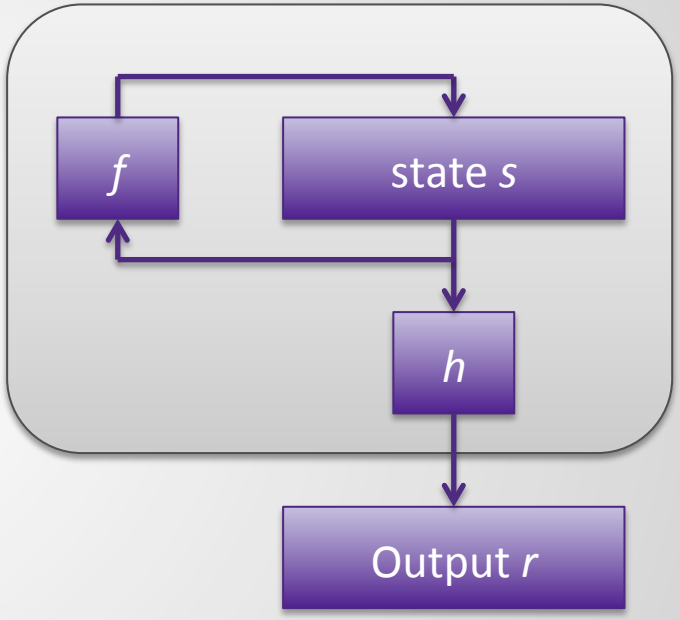
Reason 2: Unforseen/unintended problems. Recall an RSA n=pq where p,q are randomly chosen primes. If it ever happened that two people ever used the same prime, you could factor their keys:

- Let n=pq and let m=pr for random primes p,q,r. Computing the greatest common divisor yields a factor: GCD(n,m) = p.
- Researchers downloaded all RSA moduli on the internet and GCD'd them all with each other and were able to factor many keys generated by bad RNGs.

Cryptographically Secure Pseudo-random Number Generators (CSPRNG)

Anatomy of a CSPRNG

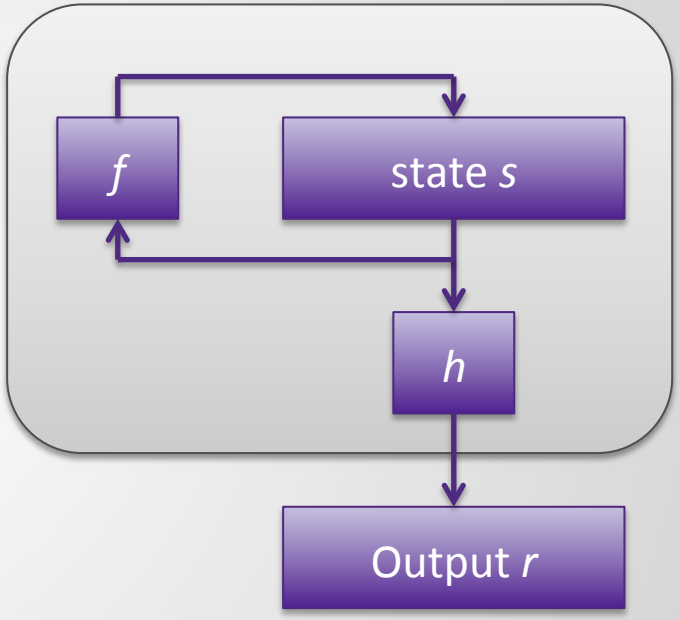
- An internal state *s*
  - Initial state s_0 is often called a *seed*
- One-way updating function *f*
  - Next state: $s_{i+1}=f(s_i)$
- One-way output function *g*
  - Output: $r_i = g(s_i)$
- Assumptions:
  - Everyone knows *f, h* as per Kecrkhoff's principle
  - Eavesdropper knows output r (e.g. from client_random)
  - State *s* is a secret, known only to user



Cryptographically Secure Pseudo-random Number Generators (CSPRNG)

Security properties:

- *h* is one-way: you shouldn't be able to guess *s* given *r*
  - *An attacker that could do this could generate all future outputs*
- *f* is one-way: you shouldn't be able to guess s_{i-1} given s_i
  - *An attacker that could do this could regenerate all past outputs*
- Should be indistinguishable from uniform random bits
  - Given previous output r, attacker shouldn't be able to guess the next bit with advantage (called the next bit test)



Example: Fortuna/CTR_DRBG

- Deterministic Random Bit Generator based on a block cipher in CTR mode
- Key and counter are drawn from an entropy accumulator
- Output function *g* is the AES encryption of state *s*.
- Updating function f is counter: $s_{i+1}=s_i+1$

Example: Fortuna/CTR_DRBG

- Hang on: Updating function *f* is a counter? That's not one-way!
- Recall reversing f allows attacker to regenerate previous outputs (e.g., previous keys!)
- Fortuna tries to deal with this by reseeding frequently
- Also block cipher in CTR mode won't output a repeat value until the counter wraps. Statically distinguishable from true randomness given enough output.
- Exercise: how much output would you need from 128-bit CTR mode to be able to distinguish it from true randomness?

Bias Example

- Bias: a deviation from a uniform distribution
- Example: bias cased by modular reduction
- Sometimes you need an RNG randint(n) that returns a number in the range 0..n
- Often only have access to an RNG rng(b) that returns b bits, i.e. number in the range $0..2^b-1$
- Could try: return rng(n.bitlength) mod n

Generating random numbers in non-power-of-2 intervals

- Creates a modulo bias:
- Numbers *n* in the range $q<n<2^b-1$ get modulo reduced into range 0..q-1
- More likely that randint(n) returns a number *less than* n/2.

Generating random numbers in non-power-of-2 intervals

- Two approaches to prevent modulo bias:
- The correct way: `do {r = rng(b) } while r > n; return r`
  - Keeps generating numbers until one is inside the correct range
  - Pro: Unbiased (if rng() is unbiased)
  - Con: May require multiple calls to rng(), "wastes" entropy
- The efficient way: return `rng(n.bitlength + margin) mod n;`
  - Generate a bunch more bits than you actually need, then modulo reduce
  - Pro: Only 1 call to rng(), wastes less entropy on average
  - Con: Still carries a slight bias, but can be made very small with only a few additional bits. For generating 256-bit numbers, NIST suggests margin=64.

Picking Appropriate Key-Lengths

Security Levels

- Size matters: as time goes on, computers get faster, cheaper, more plentiful
- Need to make attacks harder, slower
- It would be helpful to have a “security level,” that we could apply across the board to the various primitives we’ve discussed

Bits of Security

- If we say *b*-bits of security we're saying an attacker has to do 2^b operations (hashes, decryptions, etc) to succeed in their attack objective (e.g., find a collision, decrypt, etc)

Security Levels

- Goal: Given a security level *b*, what key lengths/ parameter sizes should you use for the various primitives we've discussed?
- What are the parameters of importance?
  - Differs by primitive
  - Differs even by application (e.g., hashes)

Symmetric-key Encryption

- Easiest out the various primitives to establish
- Main threat against symmetric-key (assuming underlying cipher is secure) is a brute-force attack trying all the keys
- Given security level *b*, set keylength $k \geq b$

Discrete-log based Kx/Sigs

- Integer-based discrete log systems (i.e., DH, DSA) must make solving the discrete log problem hard
  - General purpose DL solving algorithms, e.g., baby-step/giant-step algorithm, Pollard rho, etc work in $O(2^{q/2})$ operations
  - Therefore group order q must be sized to make this take no less than 2^b operations, i.e., $|q| \geq 2q$
- Integer-based DL systems also susceptible to *index calculus* attack which operates relative to size of prime modulus p, and must be sized appropriately (see NIST guidelines)

Elliptic Curve Kx/Sigs

- Index calculus attack does not apply to Elliptic curve systems (i.e., ECDHE, ECDSA)
- Just need to ensure group size is large enough to make discrete logarithm infeasible i.e., $|q| \geq 2b$

RSA Kx/Sigs

- Unlike the discrete log setting, doubling the bits of security does not imply doubling the length of n
- Grows much faster. E.g., going from 128 to 256 increases RSA modulus length 15x
  - See: https://www.nsa.gov/business/programs/elliptic_curve.shtml
- As time goes on, RSA will become more and more inefficient relative to elliptic curve crypto (currently 6:1 at 112-bit security level, but will be 64:1 at 256-bit level)
- Refer to standards documents for minimum guidelines (currently $|n| \geq 2048$ bits)

Hashes

- In general a hash should be secure against all the main attacks (collision, pre-image, 2^{nd} pre-image)
- For example, collisions are an issue with digital signatures: if you can find a collision on two different certificates, you could present one to a CA to sign, but use the other (and it would have a valid signature)
- In this case, size of hash length must be $\geq 2b$

Hashes

- If collisions are not something that an attacker could make use of, then it suffices for the hash length to be $\geq b$
- This applies to non-digital signature applications like HMAC, and key derivation functions (e.g. PBKDF2),
- Example: HMAC
  - In order for Eve to produce a collision in an HMAC (i.e., find two messages that produce the same tag), Eve would need to know the secret key to be able to compute the tags to compare them

Choosing Appropriate Key-lengths

Primitive type	Parameter of importance	Primary Attack Strategy	Necessary Length
Symmetric-key encryption	Secret key k	Brute-force	$\|k\| \geq b$ bits
Integer-based discrete log (Diffie-Hellman, DSA)	Group order q, modulus p (where p,q, prime and p=aq+1 for some a>1)	Discrete log (small q) Index calculus (small p)	$\|q\| \geq 2b$ bits $\|p\| \approx 8b$ rounded to the nearest power of 2 (see NIST guide)
Elliptic Curve discrete log (ECDH, ECDSA)	Group order q	Discrete log	$\|q\| \geq 2b$ bits
RSA	Size of modulus n=pq	Integer factorization (general number-field sieve)	Refer to NIST guide
Hash function	Hash length l	Collisions or pre-image/ second pre-image (depending on application)	$l \geq 2b$ bits for collision resistance $l \geq b$ bits if collisions are not a risk

Current minimum NIST recommended security level

- Up until 2013, NIST allowed cryptographic primitives offering an 80-bit security level
  - Example: SHA-1 is 160 bits, implying an 80-bit security level, which was ok prior to 2013
- Today NIST disallows cryptographic primitives offering security levels less than 112 bits
  - Example: SHA-1 is now disallowed for applications where collisions can impact the security (i.e., digital signatures). Other applications, like HMAC, SHA-1 is still allowed

Key-lengths at the 112-bit security level

Primitive type	Parameter of importance	Primary Attack Strategy	Necessary Length
Symmetric-key encryption	Secret key k	Brute-force	$\|k\| \geq 112$ bits
Integer-based discrete log (Diffie-Hellman, DSA)	Group order q, modulus p (where p,q, prime and $p=aq+1$ for some a>1)	Discrete log (small q) Index calculus (small p)	$\|q\| \geq 224$ bits $\|p\| \geq 2048$ bits
Elliptic Curve discrete log (ECDH, ECDSA)	Group order q	Discrete log	$\|q\| \geq 224$ bits
RSA	Size of modulus $n=pq$	Integer factorization (general number-field sieve)	n≥ 2048 bits
Hash function	Hash length l	Collisions or pre-image/ second pre-image (depending on application)	l ≥ 224 bits for collision resistance l ≥ 112 bits if collisions are not a risk