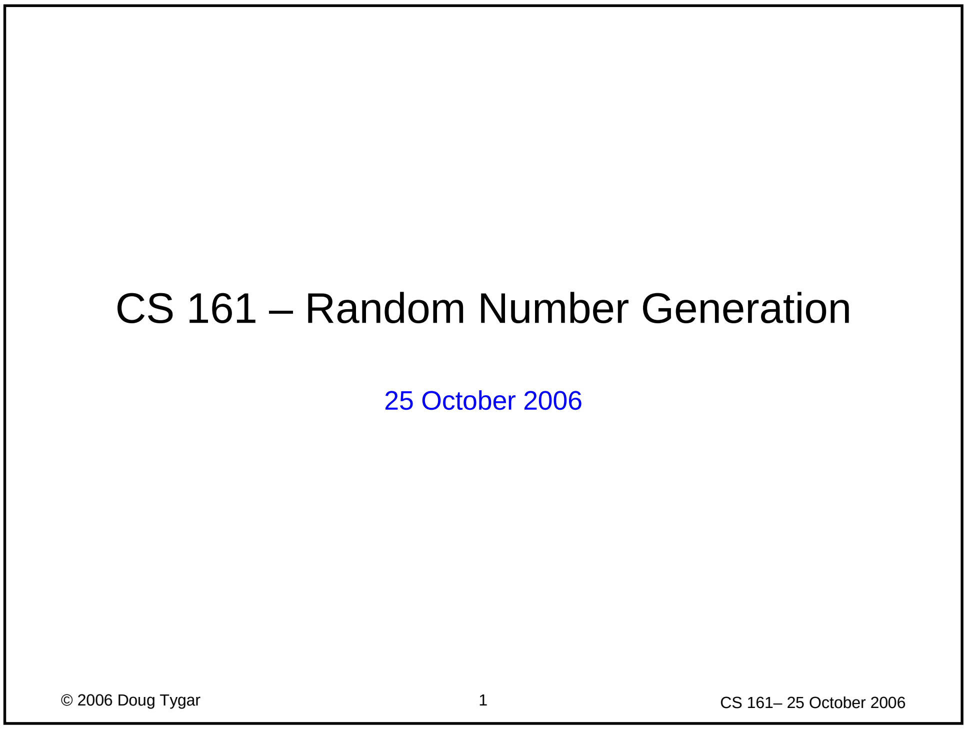
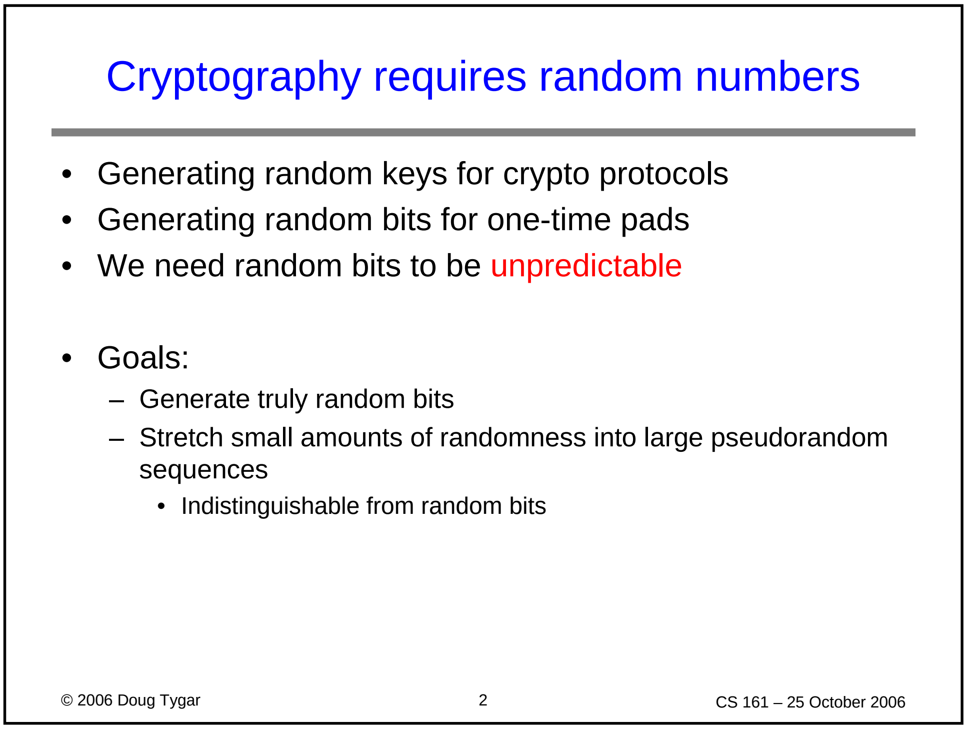


CS 161 – Random Number Generation

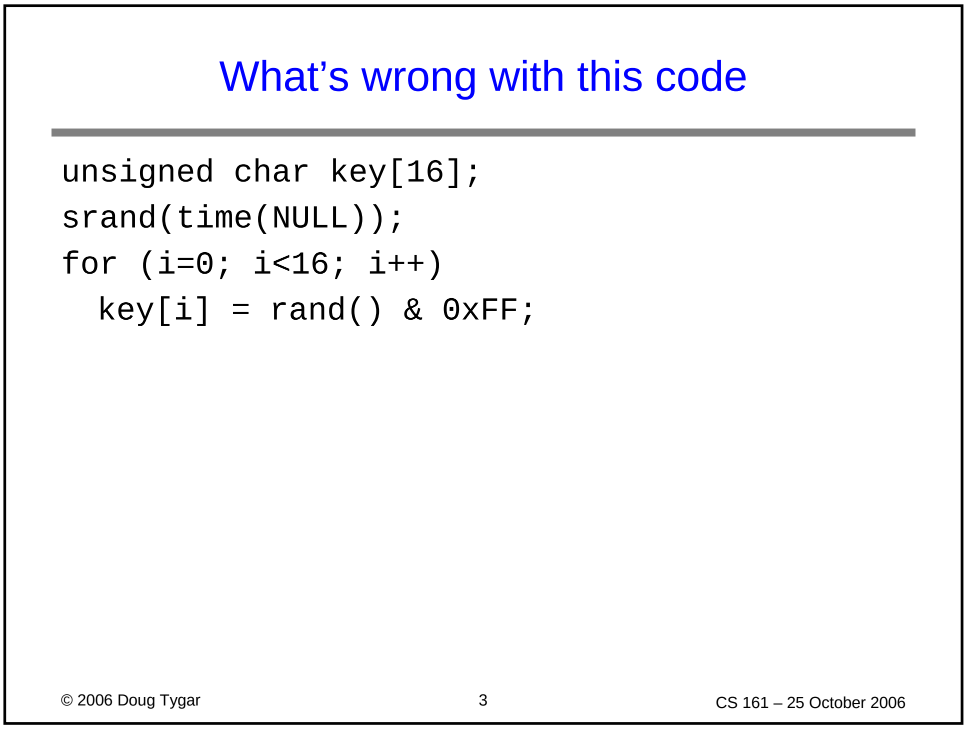
25 October 2006



Cryptography requires random numbers

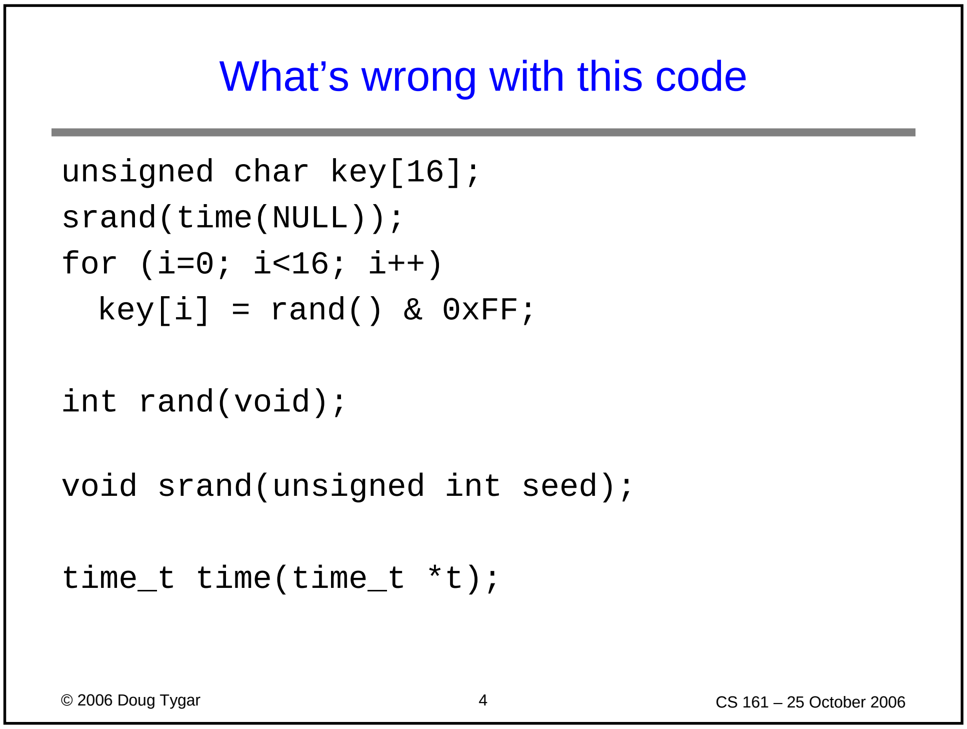
- Generating random keys for crypto protocols
- Generating random bits for one-time pads
- We need random bits to be unpredictable

- Goals:
  - – Generate truly random bits
  - – Stretch small amounts of randomness into large pseudorandom sequences
    - Indistinguishable from random bits



What’s wrong with this code

```
unsigned char key[16];
srand(time(NULL));
for (i=0; i<16; i++)
  key[i] = rand() & 0xFF;
```



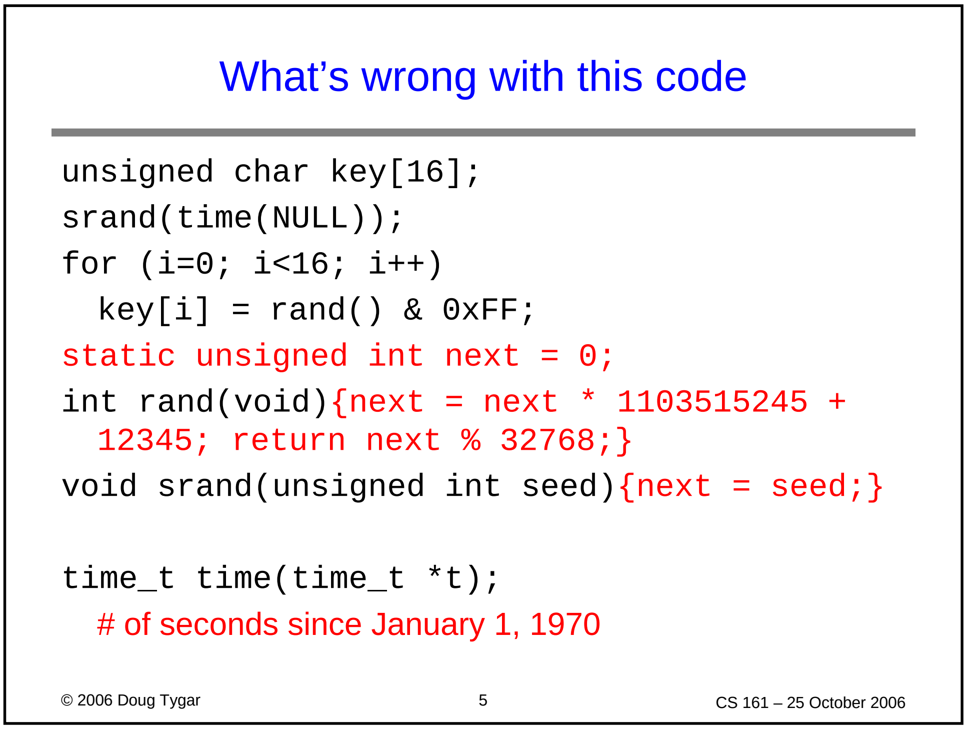
What’s wrong with this code

```
unsigned char key[16];
srand(time(NULL));
for (i=0; i<16; i++)
  key[i] = rand() & 0xFF;

int rand(void);

void srand(unsigned int seed);

time_t time(time_t *t);
```

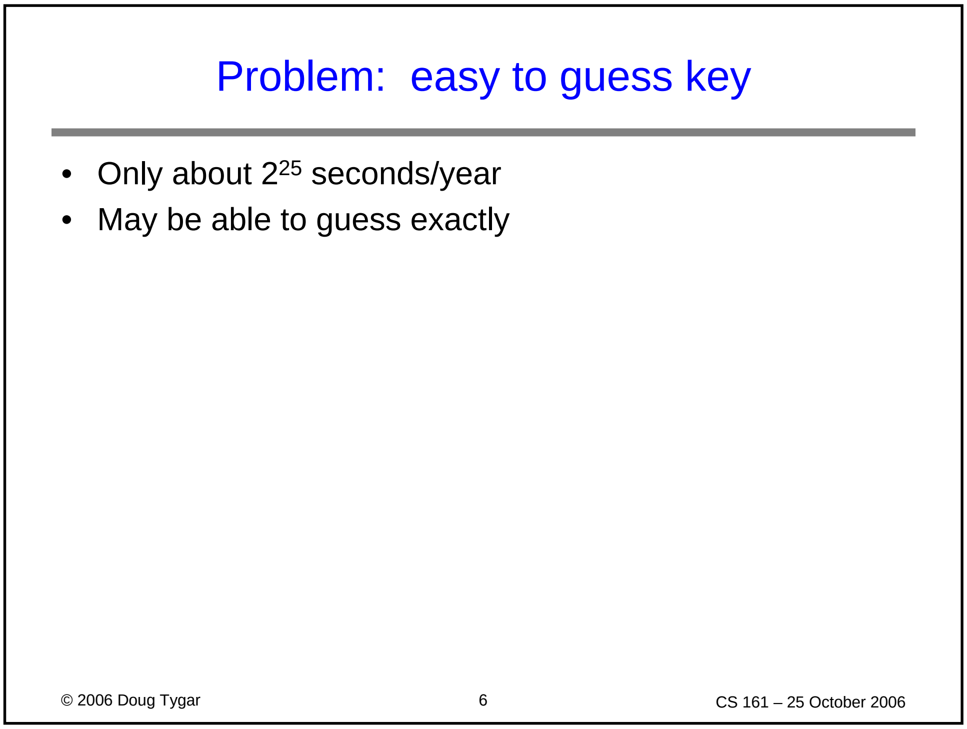


What’s wrong with this code

```
unsigned char key[16];
srand(time(NULL));
for (i=0; i<16; i++)
  key[i] = rand() & 0xFF;
static unsigned int next = 0;
int rand(void){next = next * 1103515245 +
  12345; return next % 32768;}
void srand(unsigned int seed){next = seed;}

time_t time(time_t *t);
```

# of seconds since January 1, 1970



Problem: easy to guess key

- Only about 2^{25} seconds/year
- May be able to guess exactly

Problem: Output is not random

```
int rand(void){
  next = next * 1103515245 + 12345;
  return next % 32768;
}
```

- Output is not random (low order bits flips between 0 & 1)
- Output of rand depends on previous value!

Examples of real problems

- Netscape generated SSL keys using time & process ID as seed; easily guessable & breakable
- RSA keys generated same way in Netscape
- Kerberos had same problem in generating keys
- Another Kerberos problem: `memset()` to erase seed after used actually erased seed before it was used; seed always zero
- X Windows “magic cookie” generated as shown above; only 2^8 random values

Examples of real problems

- Sun NFS filehandles generated based on pseudorandom value from time of day and process ID; this allows anyone who can guess filehandle to access file
- Similar problems in DNS resolvers
- Majordomo had bad pseudorandom number generator; could forge mailing list acceptance
- PGP used return value of `read()` (rather than read buffer) to seed generator; but `read()`always returns 1 (byes read)
- Online poker site used bad random number generator; could be guessed allowing one to always win at poker

Morals

- Seeds must be unpredictable
  - 128 bit sequences are sufficient
  - All possibilities equally likely
  - Best if seed is truly random
- Pseudorandom generator must be secure
  - No detectable pattern
  - Even if attacker can guess some pseudorandom bits, must not be able to find other pseudorandom bits

Two types of generators

- Truly random number generator (TRNG)
- Cryptographically-secure Pseudorandom number generator (CS-PRNG)

- CS-PRNG not distinguishable from truly random bits
- Distinguishing equivalent to breaking cryptosystem

Structure

- First, generate a seed
  - Truly random
  - For example, 128 bits
  - Similar to a cryptographic key
- Generate pseudorandom output based on the seed
  - Stretched into larger sequence
  - Billions of bits are no problem

CS-PRNG

- Easy to generate
- For example, we can computer AES-CBC(*seed*, 0^n) to generate n pseudorandom bits

TRNG

- One idea is to use physical process
- Use randomness from other sources
  - High-speed clock (nanosecond level)
  - Soundcard
  - Keyboard input
  - Disk timing
- We want to combine data from many sources
- Good approach: use cryptohash (e.g., SHA-1)
- What doesn’t work
  - IP address
  - IP packet content
  - Process ID