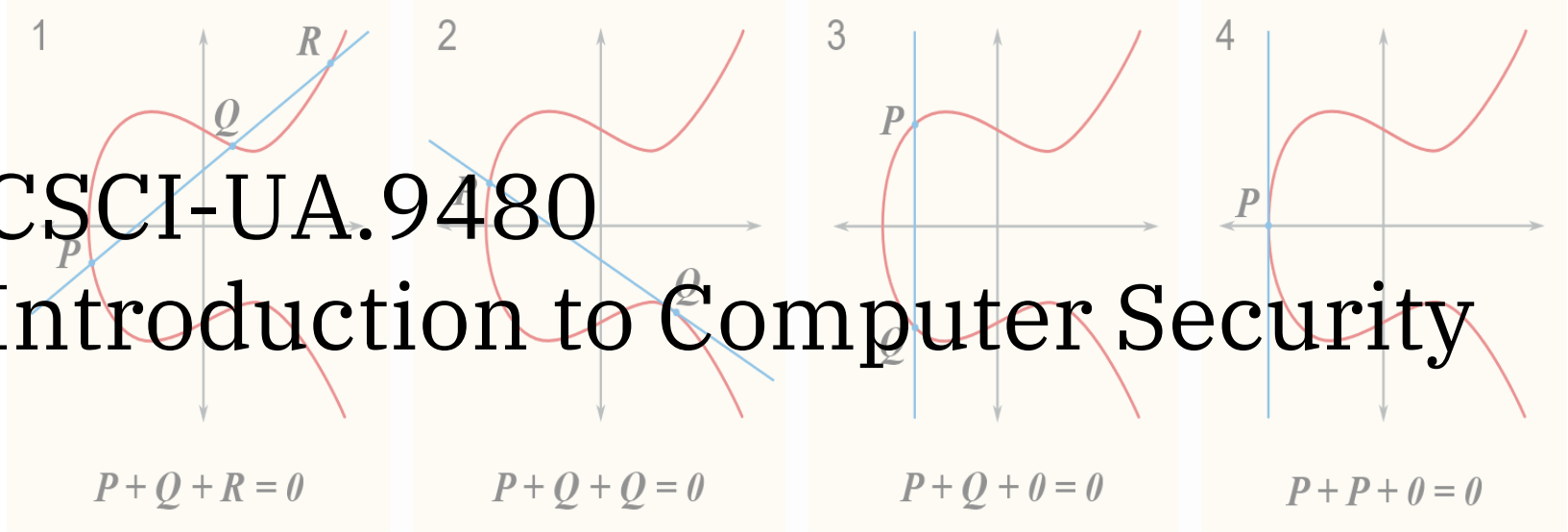


CSCI-UA.9480 Introduction to Computer Security

NYU

Session 1.3
Public Key Cryptography and Randomness

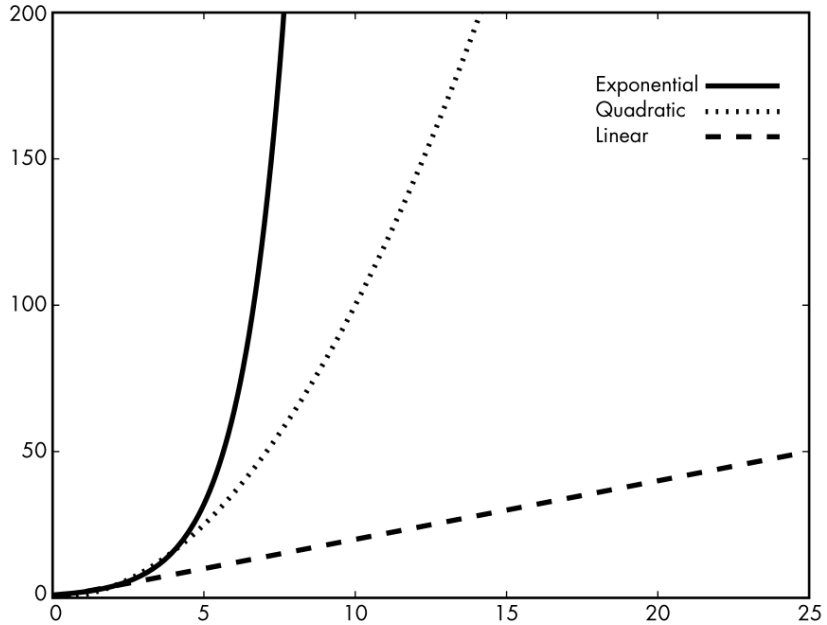
Prof. Nadim Kobeissi

Hard Problems

1.3a

Evaluating computational difficulty.

- Computational hardness can be generally evaluated using Big-O notation.
- But we also want to evaluate computational complexity:
  - **P:** Polynomial time algorithms.
  - **NP:** Nondeterministic polynomial time algorithms.



Test your knowledge!

What is the computational complexity of this search algorithm?

- ☐ **A**: $O(n)$
- ☐ **B**: $O(n^2)$
- ☐ **C**: $O(2^n)$

```
let search = (array, x) => {
    for (i = 0; i < array.length; i++) {
        if (array[i] === x) {
            return i;
        }
    }
    return -1;
}
```

Test your knowledge!

What is the computational complexity of this search algorithm?

☑ **A:** $O(n)$

☐ **B:** $O(n^2)$

☐ **C:** $O(2^n)$

```
let search = (array, x) => {
    for (i = 0; i < array.length; i++) {
        if (array[i] === x) {
            return i;
        }
    }
    return -1;
}
```

P-complete problems are solvable in polynomial time: $O(n^k)$.

NP-complete problems are problems that don’t know how to solve in polynomial time but that we can verify in polynomial time.

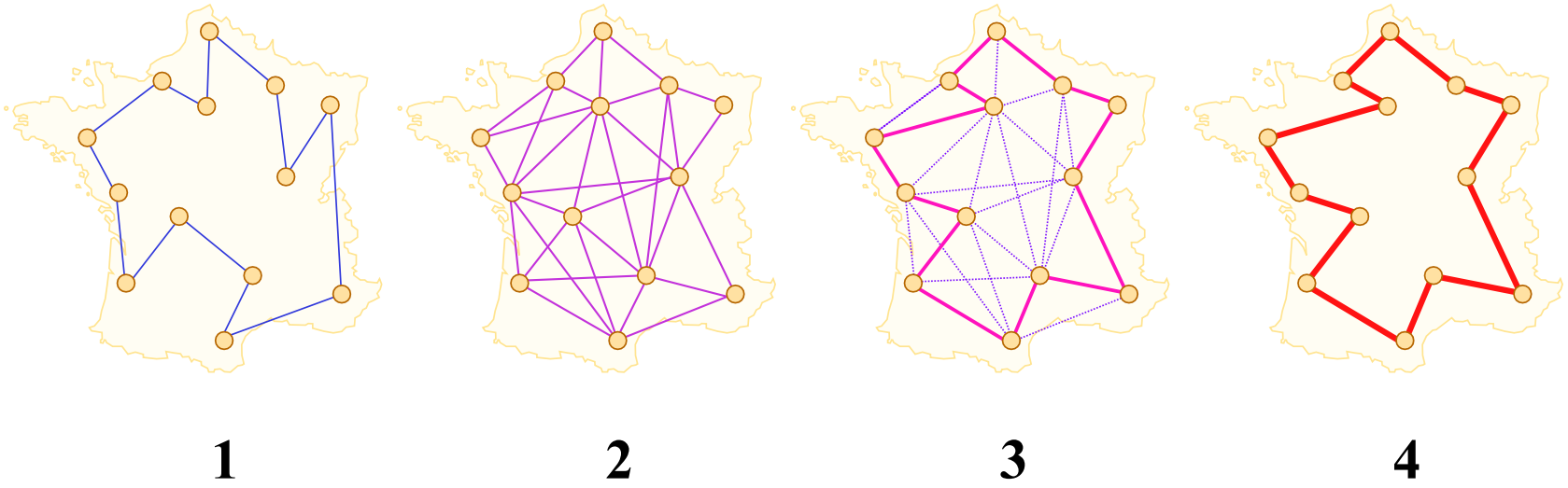
NP-complete problem: traveling salesman.

Find a path that visits every home in a city while consuming the least amount of gas.

- Solution not immediately obvious (especially for larger cities.)
- Verifying a solution is somewhat more obvious.

NP-complete problem: traveling salesman.

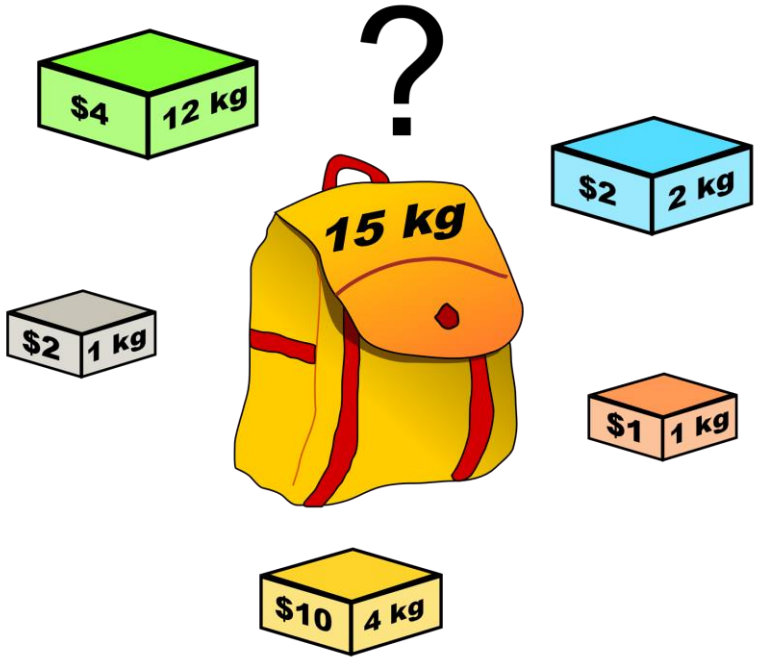
“Ant colony optimization”: quality of pheromones proportional to the efficiency/length of the path.



NP-complete problem: knapsack.

Can you find the cheapest way to fill the knapsack with 15kg of weights?

- Solution not immediately obvious (especially for much larger knapsacks.)
- Solution easily verifiable.



Did you know?

Tetris can be considered an NP-class problem: difficult to solve but with easy to verify solutions.

NP-complete problem: Tetris!

Hard to clear lines, easy to verify a replay of someone else playing.

- All NP-complete problems can be reduced to one another.
- Nobody has proven that P ≠ NP.
- But we're almost sure that hard problems *do* exist.

Link each icon to the correct label.

Hashing x to get y.

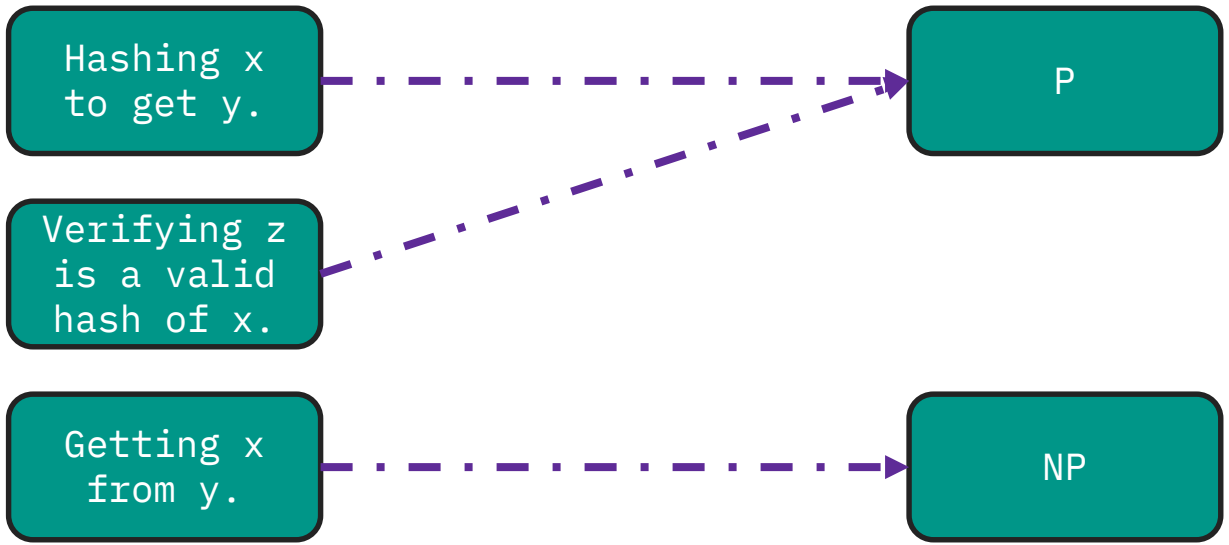
Verifying z is a valid hash of x.

Getting x from y.

P

NP

Link each icon to the correct label.



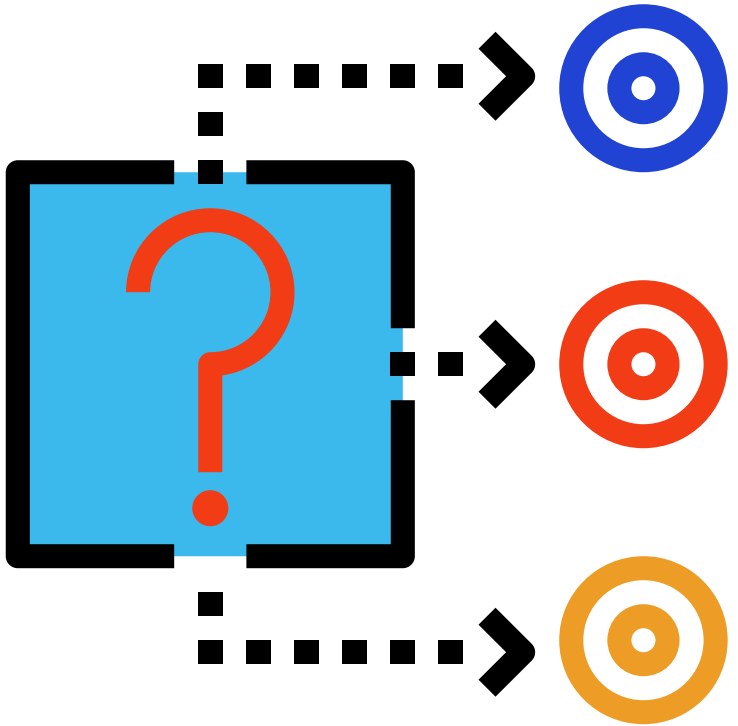
Diffie-Hellman

and Elliptic-Curve Diffie-Hellman

1.3b

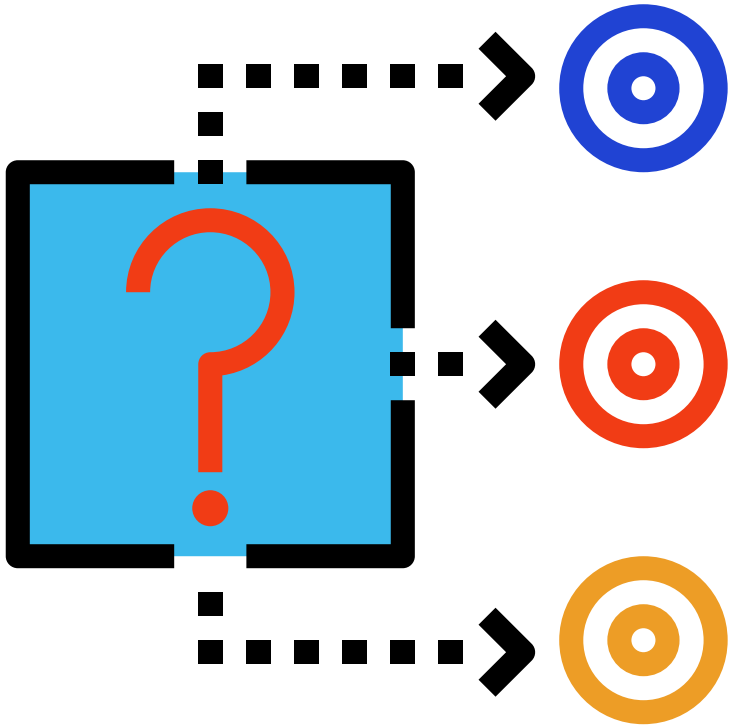
Hard problems: RSA.

- Given $N = p \times q$ where p and q are large prime numbers, can you find p and q?
- If N is a 2048-bit number, it would have two prime factors of ~1000 bits each, making it take 2^{90} operations to break.
- This is the root of the RSA public key encryption scheme.
- Other public key encryption schemes are similarly rooted in different hard problems.



Hard problems: Diffie-Hellman.

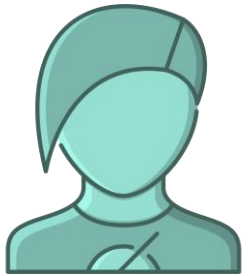
- Given $g^y = x$ where you only know g and x, can you find y?
- We operate in a group Z_p*, the set of all positive integers up until a large prime number p.
- All operations are modulo p: the group loops back on itself.



Hard problems: Diffie-Hellman.

a

g^a



b

g^b

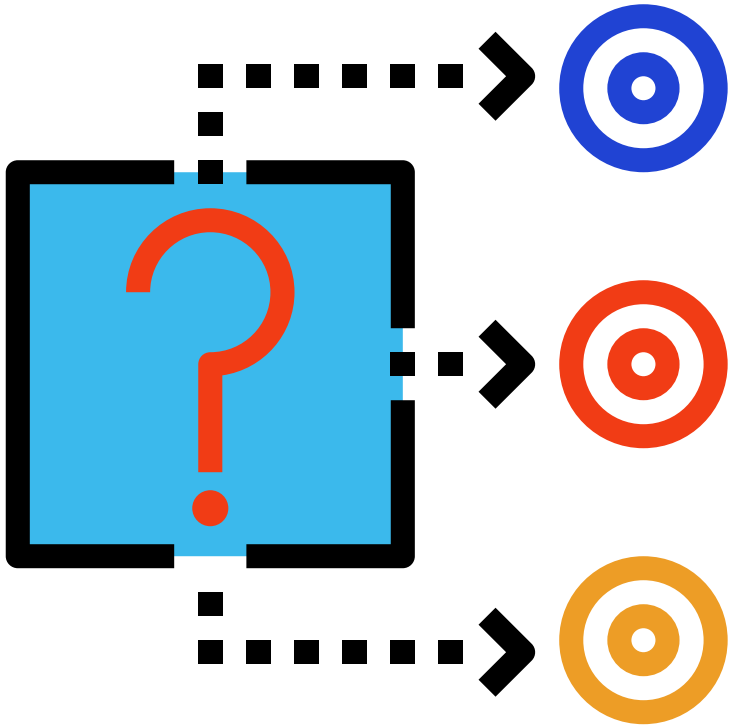
$g^a \bmod p$ →

← $g^b \bmod p$

Public values: g, p
Private keys: a, b
Public keys: g^a, g^b
Shared secret: $g^{ab} \bmod p$

Hard problems: Diffie-Hellman.

- *Computational Diffie-Hellman problem:* Given g^a and g^b, can you calculate g^{ab}?
- *Decisional Diffie-Hellman problem:* Given g^a, g^b and some value g^c for some random c, can you differentiate g^{ab} from g^c?



Attacker model for key agreement.

- *Eavesdropping*: a passive attacker listens on the network.
- *Man-in-the-middle*: an active attacker substitutes values on the networks.
- *Device compromise:* an attacker steals your smartphone.

As discussed last time: protocols.

In *protocols*, we reason about:

- Principals: Alice, Bob.
- Security goals: confidentiality, authenticity, forward secrecy...
- Use cases and constraints.
- Attacker model.
- Threat model.

As discussed last time: protocols.

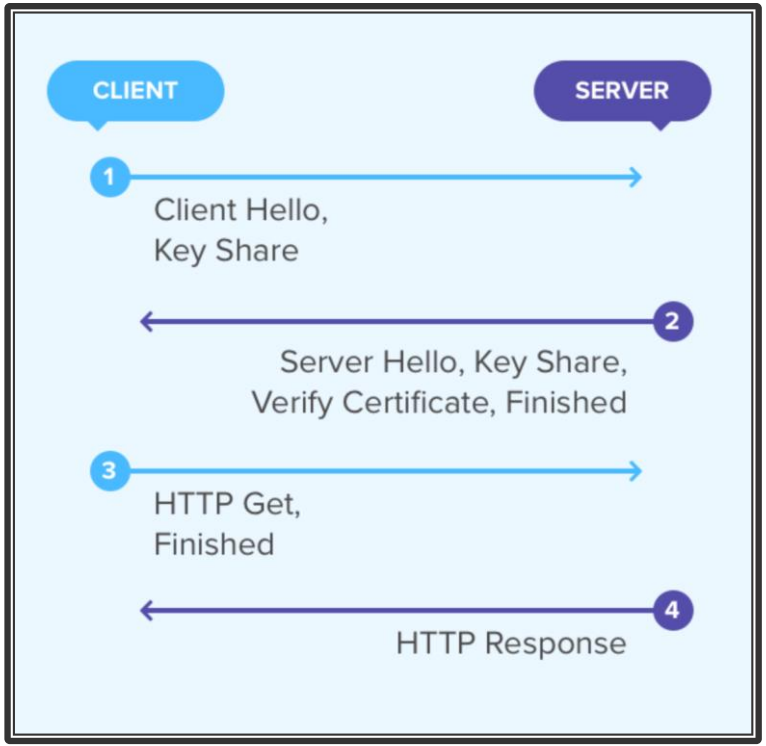
Protocols are frequently entrusted with:

- Communicating secret data without a malicious party being able to read it: *confidentiality*.
- Ensuring that any data Bob receives that appears to be from Alice is indeed from Alice: *authenticity*.
- Limiting the damage that can be caused by device compromise or theft: *post-compromise security.*

As discussed last time: protocols.

In TLS 1.3 (the latest engine for HTTPS):

- The server *authenticates* itself to the client using signed certificates.
- The client *encrypts* data to the server using ciphers and integrity codes.
- Key agreement uses Diffie-Hellman.



Elliptic curve Diffie-Hellman.

- Number field sieve algorithm makes solving the discrete logarithm in regular Diffie-Hellman groups (Z_p*) somewhat fast.
- This doesn't apply when the group is over an elliptic curve (521-bit key sizes are great.)

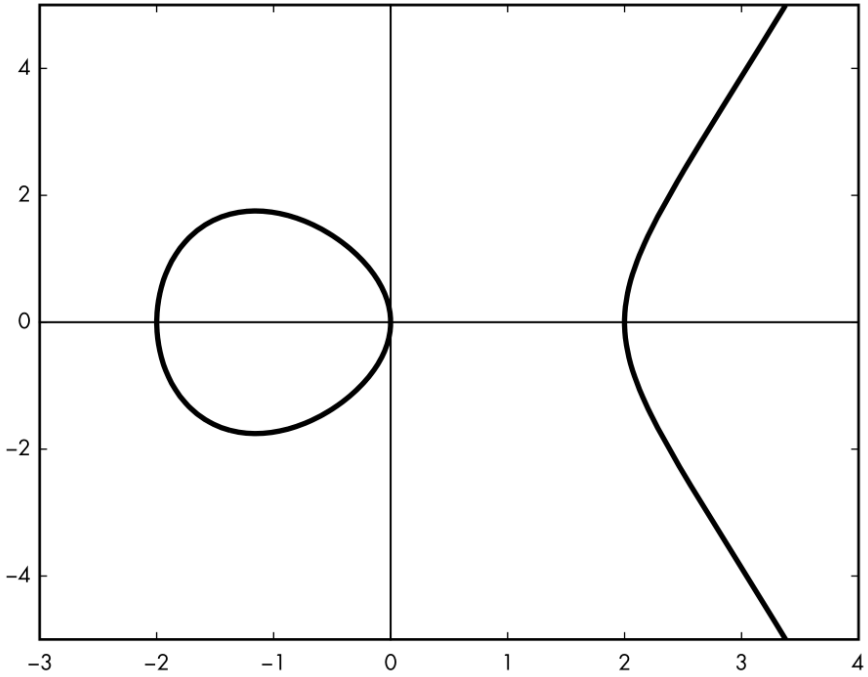


Figure 12-1: An elliptic curve with the equation $y^2 = x^3 - 4x$, shown over the real numbers

Elliptic curve Diffie-Hellman.

Elliptic curve Diffie-Hellman.

- Special rules for addition and scalar multiplication.
- “Safe curves” must be chosen: https://safecurves.cr.yp.to
- Elliptic Curve Discrete Logarithm problem is the reduction.
- EC Diffie-Hellman: X25519.
- EC Signatures: Ed25519.

		Parameters:			ECDLP security:				ECC security:			
Curve	Safe?	field	equation	base	rho	transfer	disc	rigid	ladder	twist	complete	ind
Anomalous	False	True✓	True✓	True✓	True✓	False	False	True✓	False	False	False	False
M-221	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓
E-222	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓
NIST P-224	False	True✓	True✓	True✓	True✓	True✓	True✓	False	False	False	False	False
Curve1174	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓
Curve25519	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓
BN(2,254)	False	True✓	True✓	True✓	True✓	False	False	True✓	False	False	False	False
brainpoolP256t1	False	True✓	True✓	True✓	True✓	True✓	True✓	True✓	False	False	False	False
ANSSI FRP256v1	False	True✓	True✓	True✓	True✓	True✓	True✓	False	False	False	False	False
NIST P-256	False	True✓	True✓	True✓	True✓	True✓	True✓	False	False	True✓	False	False
secp256k1	False	True✓	True✓	True✓	True✓	True✓	False	True✓	False	True✓	False	False
E-382	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓
M-383	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓
Curve383187	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓
brainpoolP384t1	False	True✓	True✓	True✓	True✓	True✓	True✓	True✓	False	True✓	False	False
NIST P-384	False	True✓	True✓	True✓	True✓	True✓	True✓	False	False	True✓	False	False
Curve41417	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓
Ed448-Goldilocks	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓
M-511	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓
E-521	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓	True✓

Signature Schemes.

Useful for attesting the integrity and authenticity of data to a wide audience without prior key agreement or secret exchange.

- Usually the slowest primitive.
- Elliptic-curve signature schemes are widely used today (RSA is on its way out.)
- Hash-based signatures exist but are slower (except if your number of safe signatures is bounded.)

What about quantum computers?

- DH, ECDH and RSA are **not** post-quantum safe. Examples of post-quantum algorithms:
  - Any hash-based signature scheme.
  - Code-based schemes.
  - Lattice-based schemes.
- Great resources on PQ cryptography:
  - *Serious Cryptography*, Chapter 14.
  - https://pqcrypto.org

Fig. 1: A fully functional, fast quantum computer.

Randomness

Following slides based on a slide deck by J.P. Aumasson and Philipp Jovanovic.

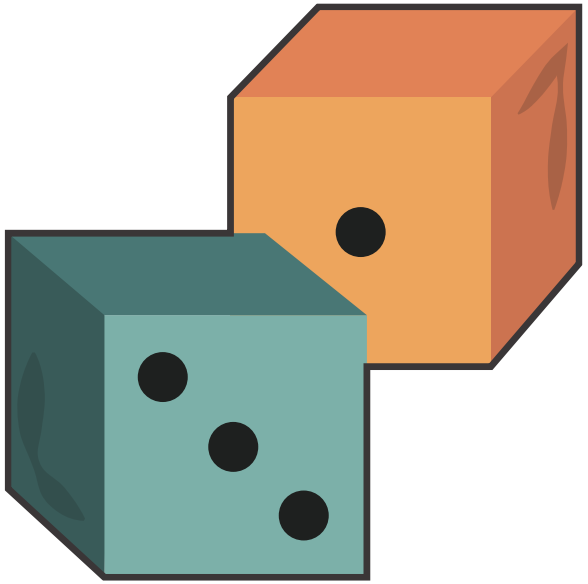
1.3c

“Random numbers are absolutely essential for a crypto library, if they’re not good enough, we don’t even have to get started with encryption or anything else, because it all collapses to something trivially deterministic and therefore predictable.”
– *Martin Boßlet*

Randomness in cryptographic systems.

Why do we need strong randomness?

- Generation of secret keys.
- Secure encryption.
- Key agreement protocols (Signal, TLS, etc.)
- Side-channel defenses.
- And other use cases.



Test your knowledge!

Have these numbers been randomly generated?

```
01001101111010111010 10
```

Test your knowledge!

Have these numbers been randomly generated?

```
01001101111010111010
```

$$Probability = 1/2^{20}$$

Test your knowledge!

Have these numbers been randomly generated?

`01001101111010111010`

$$Probability = 1/2^{20}$$

2 = number of possible bits (0, 1)

20 = number of bits in the bitstring

Test your knowledge!

Have these numbers been randomly generated?

```
00000000000000000000
```

Test your knowledge!

Have these numbers been randomly generated?

`00000000000000000000`

$$Probability = 1/2^{20}$$

2 = number of possible bits (0, 1)

20 = number of bits in the bitstring

“There is no such thing as a random number – there are only methods to produce random numbers.”
– *John von Neumann*

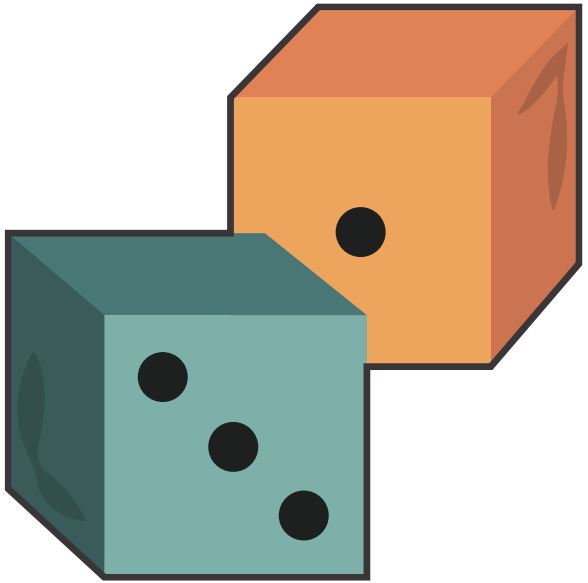
Randomness in cryptographic systems.

RNGs produce random bits.

- Non-deterministically.
- Thanks to external analog sources (waterfall, quantum measurements...)

DRBGs produce pseudorandom bits.

- Deterministically.
- From a seed (hopefully taken from an RNG or similar.)



Randomness in cryptographic systems.

RNGs produce random bits.

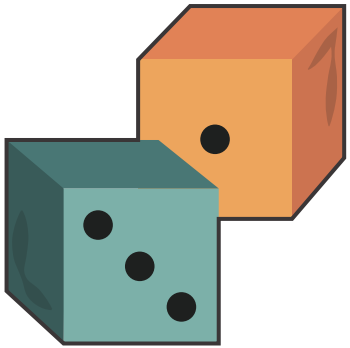
- Non-deterministically.
- Thanks to external analog sources (waterfall, quantum measurements...)

PRNGs produce pseudorandom bits.

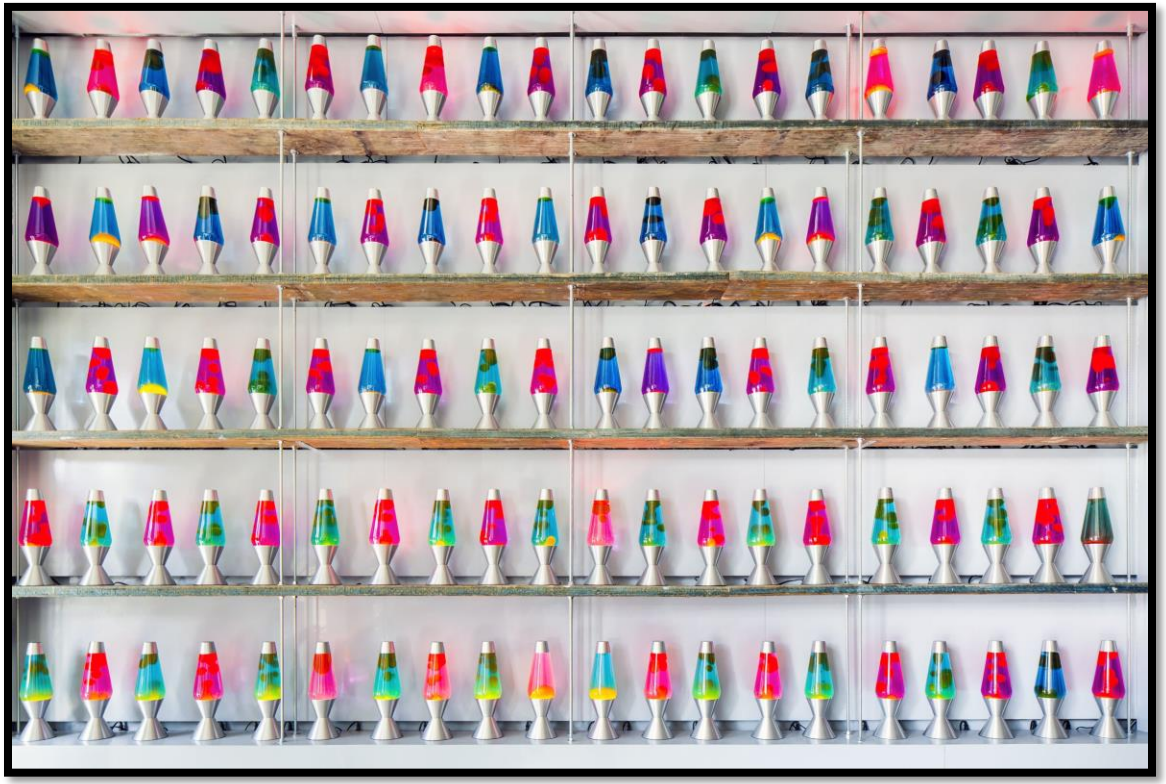
- Non-deterministically.
- Uses seeds from an RNG to maintain entropy pools.

DRBGs produce pseudorandom bits.

- Deterministically.
- From a seed (hopefully taken from an RNG or similar.)



Cloudflare uses a wall of lava lamps!

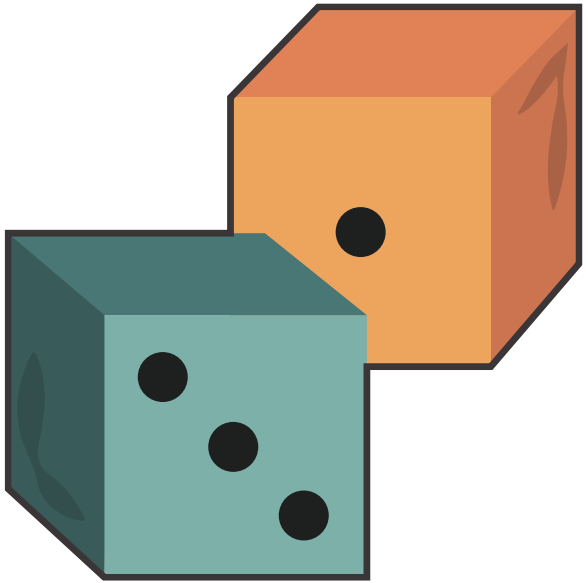


“LavaRand”:

https://blog.cloudflare.com/lavarand-in-production-the-nitty-gritty-technical-details/

Entropy: measuring uncertainty.

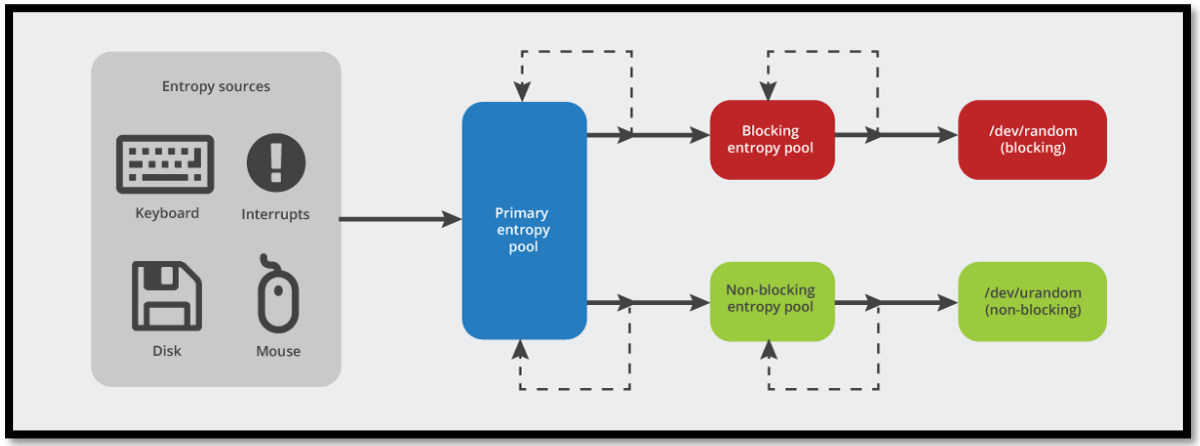
- *Symmetric keys*: entropy of a key = key size in bits.
- *Public keys*: as much entropy as $\log_2$ (number of potential choices).
- If your keys need entropy of *n* bits, you should use a PRNG with entropy at least *n* to generate these keys.



The Linux Kernel PRNGs.

- */dev/random:* device file that outputs random bytes **(blocking)**
- */dev/**u**random:* device file that outputs random bytes **(non-blocking)**

Image courtesy of Cloudflare.



Windows PRNG.

- *BCryptGenRandom():* Windows' PRNG.
- However, using a safe PRNG function is not an immediate solution, as attested by this bug in QtPass reported by Jason Donenfeld in 2017.

```
for (int i = 0; i < length; ++i) {
  int index = Util::rand() % charset.length();
  QChar nextChar = charset.at(index);
  passwd.append(nextChar);
}
```

"The problem here is that modulo will not uniformly distribute that set. The proper way to do things is to just throw away values that are out of bounds."

Secondly, and more critically, here is the implementation of `Util::rand`:

```
...
  qsrand(static_cast<uint>(QTime::currentTime().msec()));
...
int Util::rand() {
#ifdef Q_OS_WIN
  quint32 ret = 0;
  if (FAILED(BCryptGenRandom(NULL, (PUCHAR)&ret, sizeof(ret),
                             BCRYPT_USE_SYSTEM_PREFERRED_RNG)))
    return qrand();
  return ret % RAND_MAX;
#else
  return qrand();
#endif
}
```

Unfortunately, using a non-cryptographically secure random number generator like libc's `rand()` is problematic -- future outputs can be derived from knowing only a handful of past outputs -- and seeding that deterministic rng with `currentTime()->msec()` is even more dangerous. Not only is the current time a guessable/bruteforcable parameter, but the documentation for `QTime::msec` actually indicates that this is merely the "the millisecond part (0 to 999) of the time", which means there are only 1000 possibilities of generated sequences of passwords.

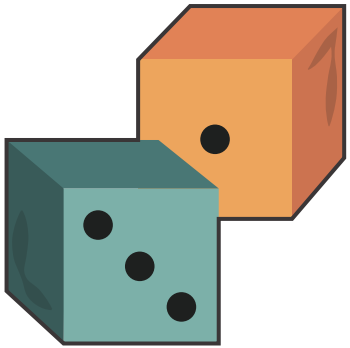
“Blocking” vs. “Non-blocking”.

`/dev/random` is blocking.

- Will freeze and stop issuing bytes (i.e. block) when entropy pool is too low.
- Entropy decreases on non-activity.

`/dev/urandom` is non-blocking.

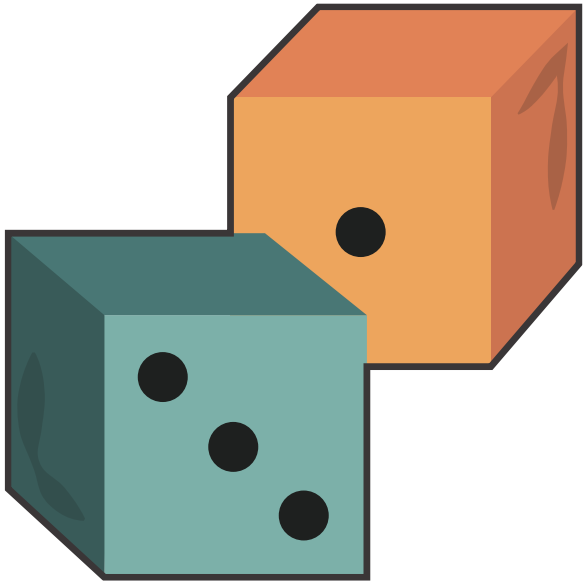
- Never freezes even when entropy pool is too low.
- Using `/dev/urandom` is perfectly fine! No need to use `/dev/random`.



What if I don’t have access to a PRNG?

If you really are stuck with no alternative, then the following (imperfect) sources can be used:

- Collect entropy from the most sources (environment, mouse movement, time, CPU temperature, system logs…)
- Hash the data collected with a secure hash function.
- Use the resulting hash to seed a strong PRNG.



Example bug: Cryptocat (2013).

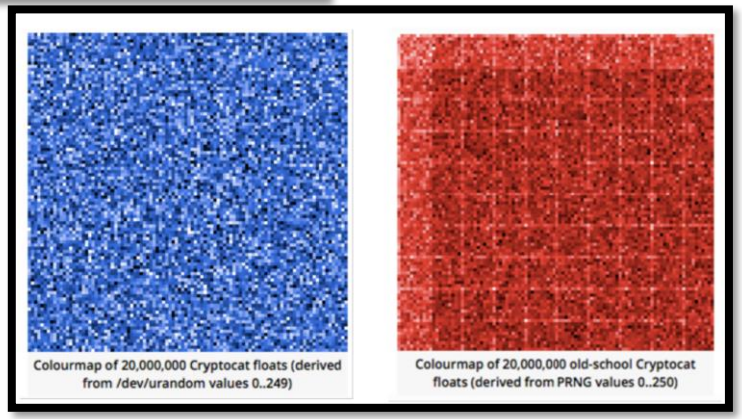
- This code is supposed to generate a string of 16 digits between 0 and 9.
- Can you identify the error?

```
Cryptocat.random = function() {
        var x, o = '';
        while (o.length < 16) {
                x = state.getBytes(1);
                if (x[0] <= 250) {
                        o += x[0] % 10;
                }
        }
        return parseFloat('0.' + o);
```

Example bug: Cryptocat (2013).

- This code is supposed to generate a string of 16 digits between 0 and 9.
- Can you identify the error?
- 25 values give a 1, 25 values give a 2…
- **26 values give a 0!**

```
Cryptocat.random = function() {
        var x, o = '';
        while (o.length < 16) {
                x = state.getBytes(1);
                if (x[0] <= 250) {
                        o += x[0] % 10;
                }
        }
        return parseFloat('0.' + o);
```



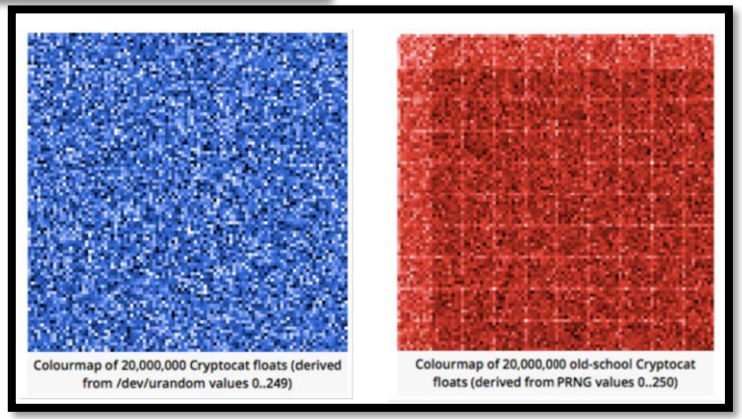
https://nakedsecurity.sophos.com/2013/07/09/anatomy-of-a-pseudorandom-number-generator-visualising-cryptocats-buggy-prng/

Example bug: Cryptocat (2013).

- This code is supposed to generate a string of 16 digits between 0 and 9.
- Can you identify the error?
- 25 values give a 1, 25 values give a 2…
- **26 values give a 0!**

16-digit string has slightly less entropy than 53 bits.

```
Cryptocat.random = function() {
        var x, o = '';
        while (o.length < 16) {
                x = state.getBytes(1);
                if (x[0] <= 250) {
                        o += x[0] % 10;
                }
        }
        return parseFloat('0.' + o);
```

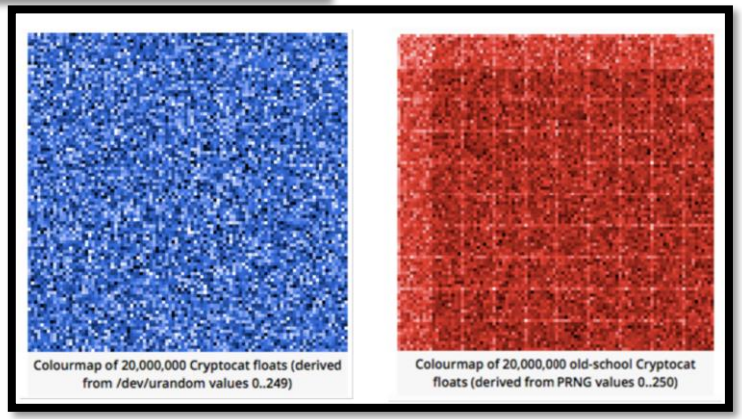


https://nakedsecurity.sophos.com/2013/07/09/anatomy-of-a-pseudorandom-number-generator-visualising-cryptocats-buggy-prng/

Example bug: Cryptocat (2013).

- This code is supposed to generate a string of 16 digits between 0 and 9.
- Can you identify the error?
- 25 values give a 1, 25 values give a 2…
- **26 values give a 0!**

```
Cryptocat.random = function() {
        var x, o = '';
        while (o.length < 16) {
                x = state.getBytes(1);
                if (x[0] <= 250) {
                        o += x[0] % 10;
                }
        }
        return parseFloat('0.' + o);
```



16-digit string has slightly less entropy than 53 bits.

Separate bug: 2^{53} isn't enough for secret keys anyway!

https://nakedsecurity.sophos.com/2013/07/09/anatomy-of-a-pseudorandom-number-generator-visualising-cryptocats-buggy-prng/

Next time: Transport Layer Security.

1.4