crypto attacks & defenses RELOADED

JP Aumasson, Philipp Jovanovic

ringzerø

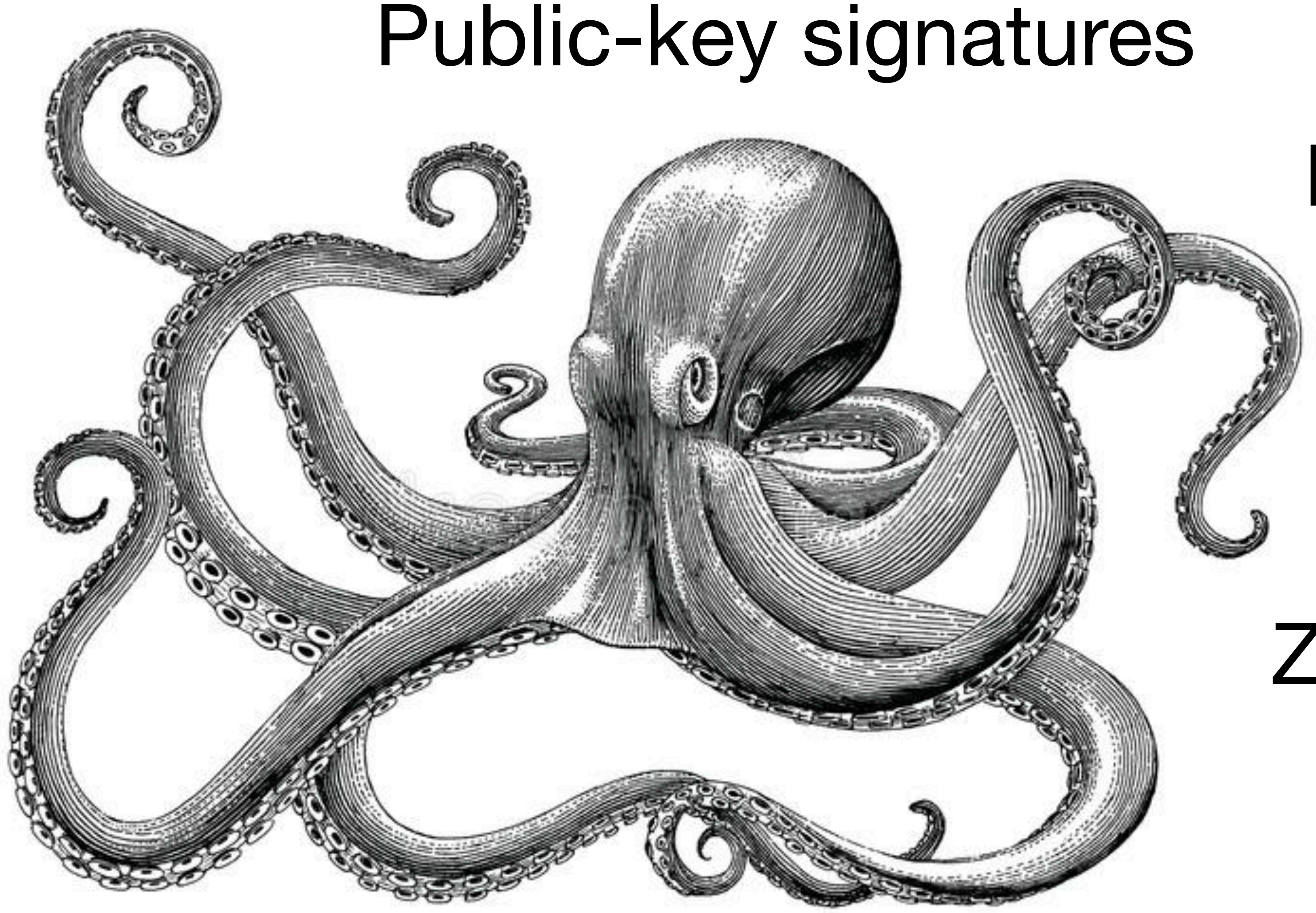
randomness

Randomness in Crypto

Key generation

Public-key signatures

Key agreement protocols



Probabilistic encryption

Zero-knowledge proofs

Side-channel defences

And many more applications!

What is Randomness

Have these bits been (pseudo)randomly generated?

010011011101011010101001100001

Looks random, no pattern, zeros and ones

Probability $(1/2)^{30}$

What is Randomness

Have these bits been (pseudo)randomly generated?

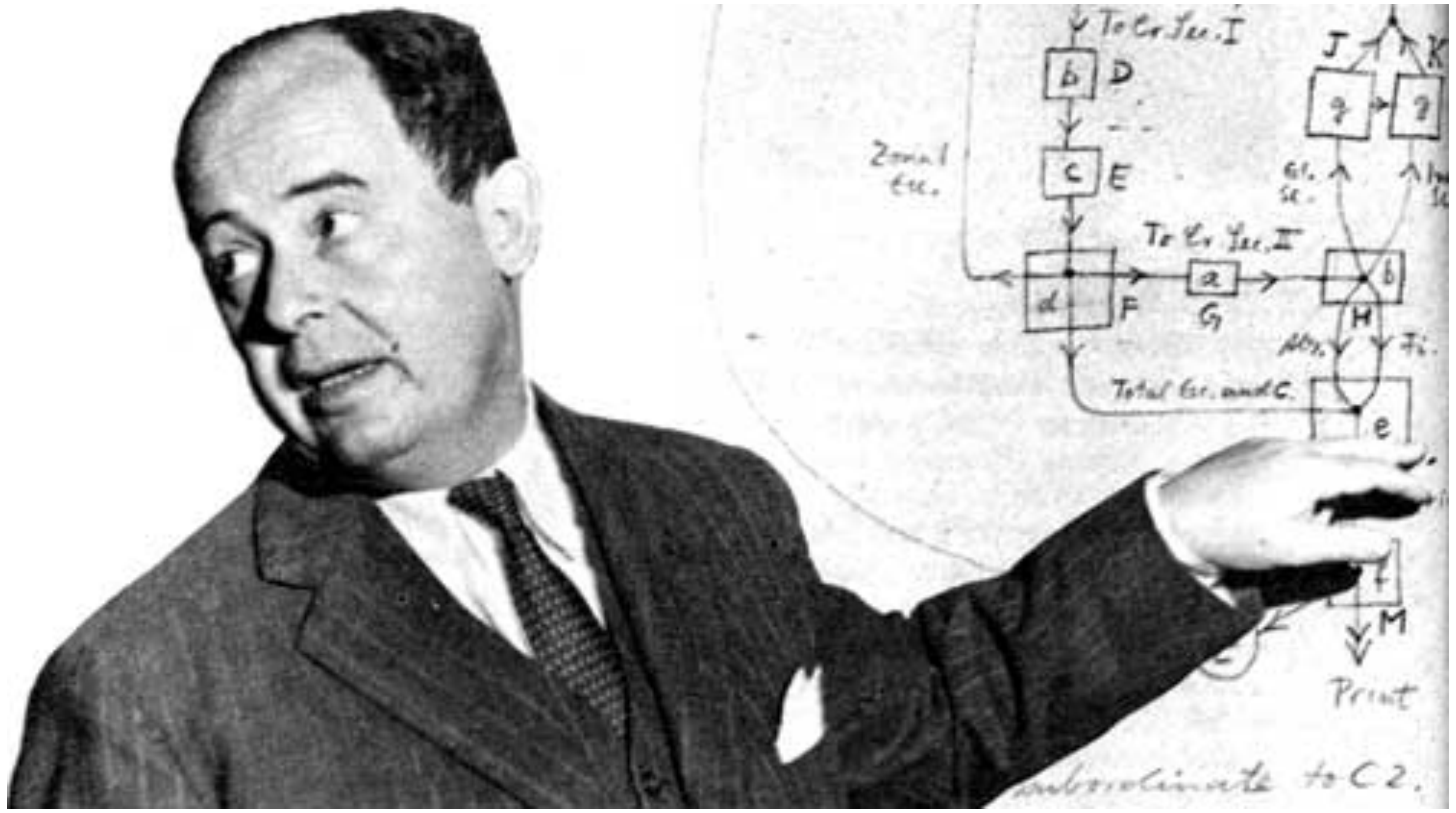
000000000000000000000000000000

Doesn't look random at all!

Same probability $(1/2)^{30}$

What is Randomness

“There is no such thing as a random number,
there are only methods to produce random numbers”



Random vs Pseudorandom

- "**True** Random" (TRNG): random, may be biased
  Not a lot of crypto, more signal processing, physical sensors

- "**Deterministic** Random Bit Generator" (DRBG): not random, not biased
  Pure crypto algorithm, taking a seed and generating a stream of bits

- “**Pseudo** Random” (PRNG): combines a TRNG and a DRBG

Today, getting good randomness is often the easiest problem in a secure system

From Random Bits to Random Objects

Or “sampling”: In many cases you will have to convert random bits/bytes into a random object: number, word combination, cards combination, etc.

There are even commercial services for this (here https://www.random.org/):

FREE services

Games and Lotteries

Lottery Quick Pick is perhaps the Internet's most popular with over 280 lotteries
Keno Quick Pick for the popular game played in many countries
Coin Flipper will give you heads or tails in many currencies
Dice Roller does exactly what it says on the tin
Playing Card Shuffler will draw cards from multiple shuffled decks
Birdie Fund Generator will create birdie holes for golf courses

Entropy: Measuring Uncertainty

If a random secret has an entropy of N bits, then (in theory) it takes at most 2^N operations to recover it

- **Symmetric keys**: entropy of a key = key size in bits
- **Public keys**: as much entropy as $\log_2$ (nb. choices)
- Generally: **entropy = $\log_2$ #choices**, if all choices are equiprobable

If your keys need entropy of N bits, you should use a PRNG with entropy at least N to generate these keys

Randomness Fail #1: Netscape (1996)

```
global variable seed;

RNG_CreateContext()
    (seconds, microseconds) = time of day; /* Time elapsed since 1970 */
    pid = process ID;  ppid = parent process ID;
    a = mklcpr(microseconds);
    b = mklcpr(pid + seconds + (ppid << 12));
     seed = MD5(a, b);

mklcpr(x) /* not cryptographically significant; shown for completeness */
    return ((0xDEECE66D * x + 0x2BBB62DC) >> 1);

MD5() /* a very good standard mixing function, source omitted */

RNG_GenerateRandomBytes()
    x = MD5(seed);
    seed = seed + 1;
    return x;
```

Randomness Fail #1: Netscape (1996)

An attacker who has an account on the UNIX machine running the Netscape browser can easily discover the pid and ppid values used in RNG_CreateContext() using the ps command (a utility that lists the process IDs of all processes on the system).

All that remains is to guess the time of day. Most popular Ethernet sniffing tools (including tcpdump) record the precise time they see each packet. Using the output from such a program, the attacker can guess the time of day on the system running the Netscape browser to within a second. It is probably possible to improve this guess significantly. This recovers the seconds variable used in the seeding process. (There may be clock skew between the attacked machine and the machine running the packet sniffer, but this is easy to detect and compensate for.)

Of the variables used to generate the seed in Figure 2 (seconds, microseconds, pid, ppid), we know the values of seconds, pid, and ppid; only the value of the microseconds variable remains unknown. However, there are only one million possible values for it, resulting in only one million

http://www.cs.berkeley.edu/~daw/papers/ddj-netscape.html

Randomness Fail #1: Netscape (1996)

Our second attack assumes the attacker does not have an account on the attacked UNIX machine, which means the pid and ppid quantities are no longer known. Nonetheless, these quantities are rather predictable, and several tricks can be used to recover them.

The unknown quantities are mixed in a way which can cancel out some of the randomness. In particular, even though the pid and ppid are 15bit quantities on most UNIX machines, the sum pid + (ppid << 12) has only 27 bits, not 30 (see Figure 2). If the value of seconds is known, a has only 20 unknown bits, and b has only 27 unknown bits. This leaves, at most, 47 bits of randomness in the secret key-a far cry from the 128-bit security claimed by the domestic U.S. version.

Randomness Fail #1: Netscape (1996)

What happened

- RNG using only weak entropy sources

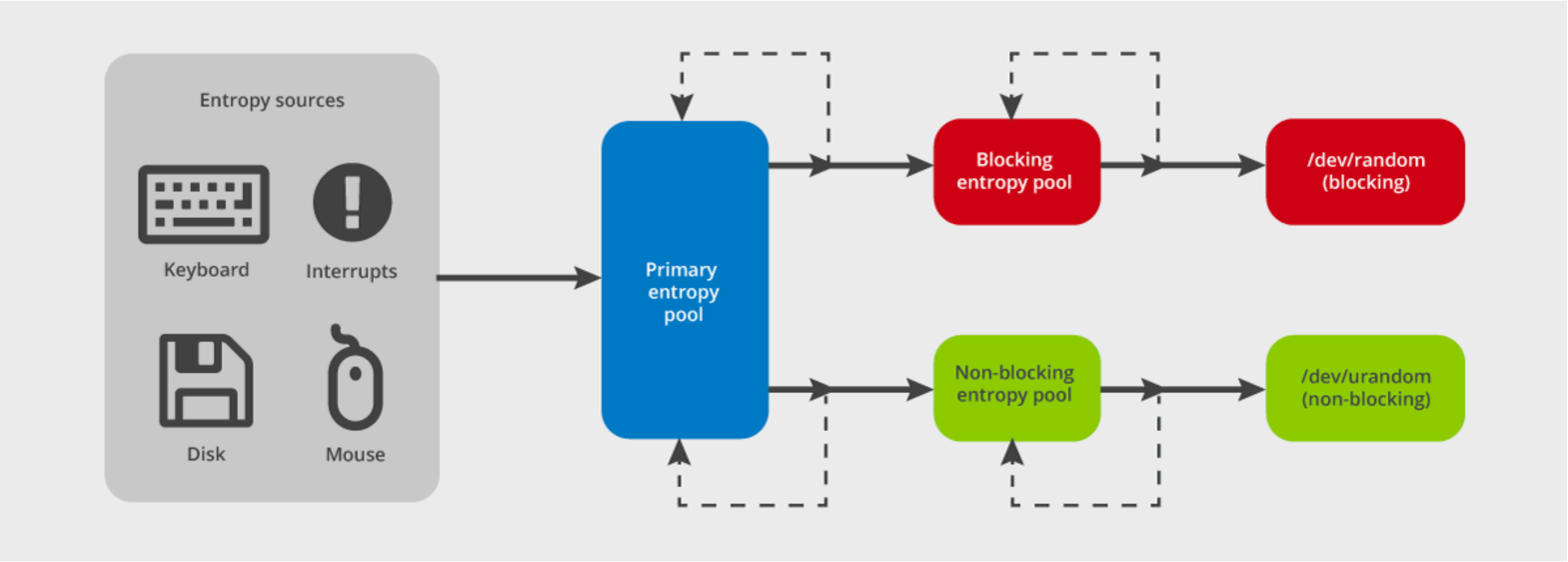
Lessons

- Don’t *only* use weak entropy sources
(IDs, timestamps, machine configuration, etc.)
- Estimate entropy
- A stronger post-processing doesn’t matter
(problem doesn’t change if MD5 is replaced with SHA-2)

The Linux Kernel PRNG (TO UPDATE!)

`/dev/random` and `/dev/urandom` device files

- 4KiB entropy pool, linear mixing, ~~SHA-1~~ BLAKE2s mixing
- Since 4.8, ChaCha stream cipher as `/dev/urandom`'s DRBG



Simple and Unsafe Use of /dev/random

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main() {
  int randint;
  int fd = open("/dev/urandom", O_RDONLY);
  if (fd != -1) {
    read(fd, &randint, sizeof randint);
  }
  printf("%08x\n", randint);
  close(fd);
  return 0;
}
```

/dev/(u)random on Linux

Current entropy in the file `/proc/sys/kernel/random/entropy_avail`

```
$ cat /proc/sys/kernel/random/entropy_avail
3459
$ dd if=/dev/random bs=1024 count=1 2>/dev/null | od -t x1 -An
 e4 f0 78 d7 21 21 ed b2 39 e1 1e ec 70 b5 a7 77
 52 bd d6 04 85 c9 0c 48 78 d3 b0 71 ef 1c a1 f6
 c6 f2 dc 58 4b 76 cf 1f 61 97 ba 50 26 58 5b ad
 5f fa 95 21 df 53 85 26 a0 90 ce f6 af 08 cd b2
 df 4b bf 3e c9 f7 99 10 55 e2 ec e4 32 c7 88 08
 09 73 8f d1 80 d2 f7 1e 3e db f1 a2 64 15 ea d0
 d1 7b 50 45 64 18 71 88 12 24 5d f4 1a ee 94 70
 7d 34 31 29 8a cb 2f a3 2e 7a b7 d6 89 76 3a b3
$ cat /proc/sys/kernel/random/entropy_avail
2216
$ cat /proc/sys/kernel/random/entropy_avail
2112
$ cat /proc/sys/kernel/random/entropy_avail
2005
```

/dev/(u)random on Linux

- `/dev/random` historically blocks when "insufficient" entropy
- Why (for example) `gpg --gen-key` complains

```
$ cat /proc/sys/kernel/random/entropy_avail
644
$ dd if=/dev/random bs=1024 count=1 2>/dev/null | od -t x1 -An
 4f cd fc f9 45 63 44 db 0e b8 02 e4 a6 2a 93 ef
 68 73 a1 cf cd 7c 43 87 9f ee 4c 52 60 77 d8 59
 af fb 06 4f e9 0c 9d 67 6d cd 16 68 88 f6 c0 01
 ef 96 13 25
$ cat /proc/sys/kernel/random/entropy_avail
29
$ dd if=/dev/random bs=1024 count=1 2>/dev/null | od -t x1 -An
 d7 ec 27 75 61 17 81 02
$ ^C
```

Attempting to dump 1KB blocks from `/dev/random`

Kernel 2022 developments

`/dev/urandom` always safe

max `entropy_avail` = 256

```
This patch goes a long way toward eliminating a long overdue userspace
crypto footgun. After several decades of endless user confusion, we will
finally be able to say, "use any single one of our random interfaces and
you'll be fine. They're all the same. It doesn't matter." And that, I
think, is really something. Finally all of those blog posts and
disagreeing forums and contradictory articles will all become correct
about whatever they happened to recommend, and along with it, a whole
class of vulnerabilities eliminated.
```

https://lore.kernel.org/lkml/20220217162848.303601-1-Jason@zx2c4.com/

Random number generator enhancements for Linux 5.17 and 5.18

by Jason A. Donenfeld (zx2c4), 2022-03-18

The random number generator has undergone a few important changes for Linux 5.17 and 5.18, in an attempt to modernize both the code and the cryptography used. The smaller part of these will be released with 5.17 on Sunday, while the larger part will be merged into 5.18 on Monday, which should receive its first release candidate in a few weeks and a release in a few months.

https://www.zx2c4.com/projects/linux-rng-5.17-5.18/

The Right Thing: getrandom()

- System call available since 3.17 (2016)
- Blocks until entropy hasn't reached an acceptable level
- Then never blocks, because entropy won't "decrease"

```
GETRANDOM(2)              Linux Programmer's Manual              GETRANDOM(2)

NAME       top

       getrandom - obtain a series of random bytes

SYNOPSIS       top

       #include <linux/random.h>

       int getrandom(void *buf, size_t buflen, unsigned int flags);
```

On Windows: BcryptGenRandom()

- Win API's crypto-secure PRNG
- Supersedes more complex CryptGenRandom() in Crypto API

The **BCryptGenRandom** function generates a random number.

Syntax

C++

```
NTSTATUS WINAPI BCryptGenRandom(
  _Inout_ BCRYPT_ALG_HANDLE hAlgorithm,
  _Inout_ PUCHAR            pbBuffer,
  _In_    ULONG             cbBuffer,
  _In_    ULONG             dwFlags
);
```

```
// Generate the AES session key.
hr = BCryptGenRandom(
     NULL,
     (BYTE*)&AesKey,
     sizeof(AesKey),
     BCRYPT_USE_SYSTEM_PREFERRED_RNG
     );

if (FAILED(hr))
{
     goto done;
}
```

Watch Out

Do not confuse *cryptographic* with *regular* **random generators**
(as used for simulations, statistics, image generation, etc.)

What PRNG Should I Use?

In C(++), either directly `getrandom()` or `BCryptGenRandom()`, or

- A library's random generator, such as OpenSSL's `RAND_bytes()`
- A crypto API that uses a library back-end (such at Themis' using BoringSSL)

In **Python/Ruby/Perl/Go/Node**/etc.: a crypto-safe randomness module

For example in **JavaScript**:

```
> var buf = new Uint8Array(8)
<· undefined
> crypto.getRandomValues(buf)
<· ▶Uint8Array(8) [165, 48, 250, 66, 20, 4, 35, 211]
> crypto.getRandomValues(buf)
<· ▶Uint8Array(8) [38, 167, 58, 42, 96, 16, 125, 44]
>
```

What PRNG Should I Use?

In the **Go** language:

```go
package main

import (
	"crypto/rand"
	"fmt"
)

func main() {

	buf := make([]byte, 8)

	_, err := rand.Read(buf)
	if err != nil {
		panic("random read failed")
	}

	fmt.Println(buf)
}
```

```go
func Read(b []byte) (n int, err error) {
	return io.ReadFull(Reader, b)
}

```

```go
// Reader is a global, shared instance of a cryptographically
// secure random number generator.
//
// On Linux, Reader uses getrandom(2) if available, /dev/urandom otherwise.
// On OpenBSD, Reader uses getentropy(2).
// On other Unix-like systems, Reader reads from /dev/urandom.
// On Windows systems, Reader uses the CryptGenRandom API.
// On Wasm, Reader uses the Web Crypto API.
var Reader io.Reader

```

PRNG Cheat Sheet

Bad	Good
rand(3)	/dev/urandom
random(3)	Linux’ getrandom()
PHP’s rand()	Java’s SecureRandom()
Mersenne Twister	Node.js’ crypto.randomBytes()
Your own DRBG / PRNG	Window’s BCryptGenRandom()
	Go’s crypto/rand
	OpenSSL’s RAND_BYTES()

Randomness Fail #2: Mediawiki (2012)

```
/**
 * Return a random password. Sourced from mt_rand, so it's not particularly secure.
 * @todo hash random numbers to improve security, like generateToken()
 *
 * @return \string New random password
 */
static function randomPassword() {
        global $wgMinimalPasswordLength;
        $pwchars = 'ABCDEFGHJKLMNPQRSTUVWXYZabcdefghjkmnpqrstuvwxyz';
        $l = strlen( $pwchars ) - 1;

        $pwlength = max( 7, $wgMinimalPasswordLength );
        $digit = mt_rand( 0, $pwlength - 1 );
        $np = '';
        for ( $i = 0; $i < $pwlength; $i++ ) {
                $np .= $i == $digit ? chr( mt_rand( 48, 57 ) ) : $pwchars{ mt_rand( 0, $l ) };
        }
        return $np;
}
```

Randomness Fail #2: Mediawiki (2012)

`mt_srand(seed)/mt_rand(min, max)`: `mt_rand` is the interface for the Mersenne Twister (MT) generator [15] in the PHP system. In order to be compatible with the 31 bit output of `rand()`, the LSB of the MT function is discarded. The function takes two optional arguments which map the 31 bit number to the [`min`, `max`] range. The `mt_srand()` function is used to seed the MT generator with the 32 bit value `seed`; if no seed is provided then the seed is provided by the PHP system.

$$x_{k+n} = x_{k+m} \oplus ((x_k \wedge 0x80000000)|(x_{k+1} \wedge 0\text{x7fffffff}))A$$

(Argyros/Kiayias, 2012)

$$xA = \begin{cases} (x \gg 1) & \text{if } x^{31} = 0 \\ (x \gg 1) \oplus a & \text{if } x^{31} = 1 \end{cases}$$

19937-bit state, but fully linear update and 32-bit seed:

Randomness Fail #2: Mediawiki (2012)

A session identifier preimage completely determines the seed of the mt_rand() and rand() PRNGs!

(Argyros/Kiayias, 2012)

⇒ Weak RNG can be exploited to

- Hijack sessions
- Predict temporary passwords

Randomness Fail #2: Mediawiki (2012)

What happened

- Weak RNG used for security purposes

Lessons

- Avoid non-crypto PRNGs, even for other applications than key generation
- In PHP, better use `openssl_random_pseudo_bytes`
  (and check the `$crypto_strong` flag)
- Don’t use PHP’s `rand()` or `mt_rand()`; C’s `random(3)`, `rand(3)`; LFSRs; etc.

Checking PRNG Security

RTFM: is it cryptographically secure?

Statistical tests

- Will find statistical biases, not crypto flaws!
- Simple tool: Ent (www.fourmilab.ch/random)

What is the **entropy source?**

- How much bits should be expected? Does if fail securely?
- Is the entropy quality consistent across OSs/platform?

Are random bits **used safely**?

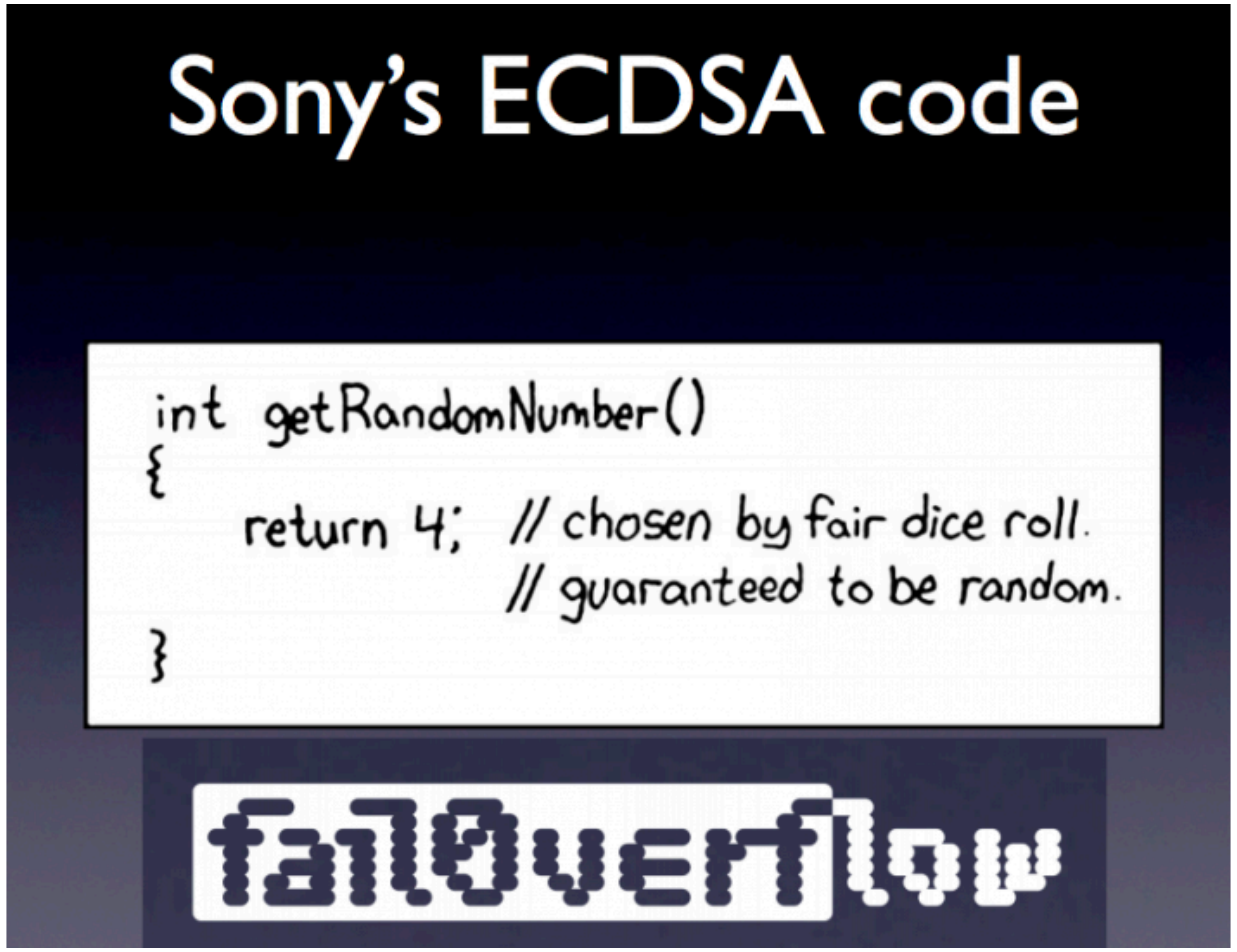
- Is sampling uniform across all possible outcomes?
- Is there enough entropy for this distribution?

Randomness Fail #3: PS3 (2010)

Sony PS3 Security Broken

Sony used an ECDSA signature scheme to protect the PS3. Trouble is, they didn't pay sufficient attention to their random number generator.

EDITED TO ADD (1/13): More info.



crypto attacks & defenses RELOADED

JP Aumasson, Philipp Jovanovic

ringzerø

randomness ✔