How random is Random?

Pitfalls of Random in .NET 6

Stan Drapkin

October 2021

Who am I?

Stan Drapkin – sdrapkin@sdprime.com

- Senior Director - Cloud Technologist - EPAM Systems Inc.
- 20 years of .NET experience
- Specialize in .NET, SQL Server, Security, Cryptography, Cloud
- OSS library author – github.com/sdrapkin
- Book author
  - Security Driven .NET (2014)
  - Application Security in .NET, Succinctly (2017)
- Conference speaker

Opinions expressed are my own

Random History 2001~2021

- .NET is 20 years old, and so is `System.Random` (SR)
- SR is a core feature of .NET, like all other members of “System” namespace

Random History 2001~2021

- .NET is 20 years old, and so is `System.Random` (SR)
- SR is a core feature of .NET, like all other members of “System” namespace
- SR has 2 constructors:
  - `Random()` => non-reproducible output (“random” seed)
  - `Random(int Seed)` => reproducible output (stable output per seed)

Random History 2001~2021

- .NET is 20 years old, and so is `System.Random` (SR)
- SR is a core feature of .NET, like all other members of “System” namespace
- SR has 2 constructors:
  - `Random()` => non-reproducible output (“random” seed)
  - `Random(int Seed)` => reproducible output (stable output per seed)
- SR docs have added the following, many years later:

“Random objects in processes running under different versions of the .NET Framework may return different series of random numbers even if they're instantiated with identical seed values.”

Random History 2001~2021

- .NET is 20 years old, and so is `System.Random` (SR)
- SR is a core feature of .NET, like all other members of “System” namespace
- SR has 2 constructors:
  - `Random()` => non-reproducible output (“random” seed)
  - `Random(int Seed)` => reproducible output (stable output per seed)
- SR docs have added the following, many years later:

  “Random objects in processes running under different versions of the .NET Framework may return different series of random numbers even if they're instantiated with identical seed values.”

- A lot of .NET code already relied on reproducible seeded SR values
- It was often bad code, but users don’t like to be told they are the problem

Random History 2001~2021

- SR actually broke seeded stability from .NET 1.1 to .NET 3.5+
- Folks complained…

Random History 2001~2021

- SR actually broke seeded stability from .NET 1.1 to .NET 3.5+
- Folks complained…
- Then got used to the new seeded output… (no one RTFM, of course)
- Then wrote 100x more code relying on and cementing the .NET 3.5 algorithm

Random History 2001~2021

- SR actually broke seeded stability from .NET 1.1 to .NET 3.5+
- Folks complained…
- Then got used to the new seeded output… (no one RTFM, of course)
- Then wrote 100x more code relying on and cementing the .NET 3.5 algorithm

This is where we are today. 2 constructors split SR into 2 worlds:

1. `Random()` => MS is free to create/improve “randomness” however it wants
2. `Random(int Seed)` => **FOREVER** stuck with 20-year-old implementation

Random History 2001~2021

- SR actually broke seeded stability from .NET 1.1 to .NET 3.5+
- Folks complained…
- Then got used to the new seeded output… (no one RTFM, of course)
- Then wrote 100x more code relying on and cementing the .NET 3.5 algorithm

This is where we are today. 2 constructors split SR into 2 worlds:

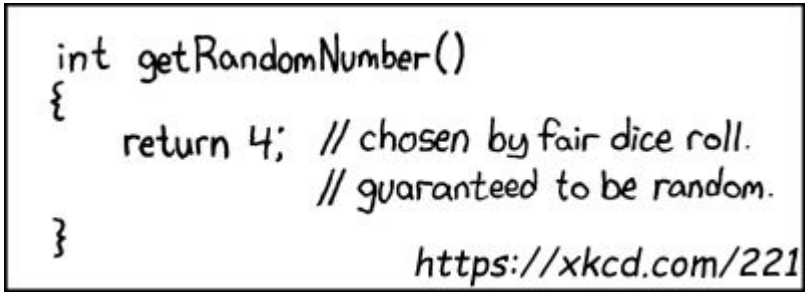
1. `Random()` => MS is free to create/improve “randomness” however it wants
2. `Random(int Seed)` => **FOREVER** stuck with 20-year-old implementation

- Stability is a feature: MS chose seeded stability over correctness
- But what’s wrong with existing `System.Random`? **DEMO TIME**

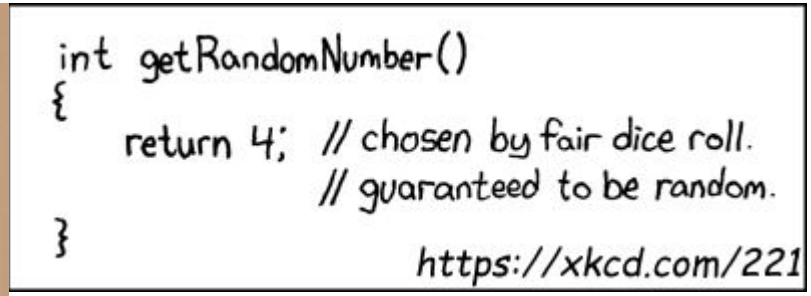
new `System.Random()` for .NET 6 ?

we need an algorithm…

Obligatory – xkcd random



Obligatory – xkcd random



LET'S SCIENCE THE ____ OUT OF THIS…

Obligatory – xkcd random

LET'S SCIENCE THE ____ OUT OF THIS…

```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

https://xkcd.com/221

Linear **C**ongruential **G**enerator (**LCG**):

$X_{i+1} = (A * X_i + C) \bmod M \qquad i = 0, 1, 2, \ldots$

Obligatory – xkcd random

LET'S SCIENCE THE ____ OUT OF THIS…

```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

https://xkcd.com/221

Linear **C**ongruential **G**enerator (**LCG**):

$X_{i+1} = (A * X_i + C) \bmod M$ $\quad i = 0, 1, 2, \ldots$

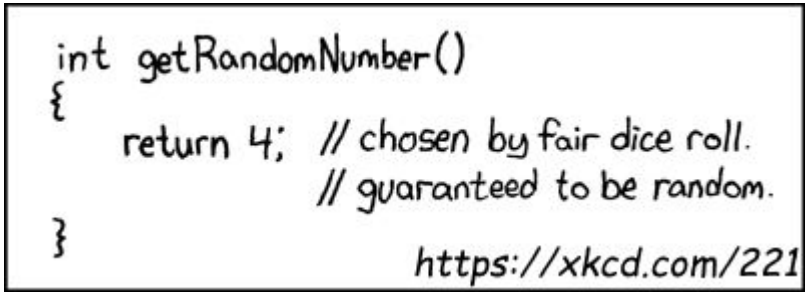
$X_{i+1} = (1 * X_i + 0) \bmod 5$ $\quad X_0 = 4$

M = modulus	= 5	$(M > 0)$
C = increment	= 0	$(C < M)$
A = multiplier	= 1	$(A < M)$
X_0 = first value	= 4	

Obligatory – xkcd random

- **Super fast!** (benchmark crashed dividing by 0)
- Vectorization friendly (SSE/AVX/SIMD)
- Tiny 32-bit global state, no per-instance state
- Thread-safe!
- Equidistributed in every dimension, no gaps
- Covers its entire output range
- No run-ups or run-downs
- All output permutations are equally likely
- Precise mathematically-proven period
- Nothing-up-my-sleeve design
- Well implemented & documented
- Consistent on all .NET runtimes and CPU arch.
- Public domain, patent-free (I hope)

What's not to like? It's **very fast**, might be **good enough**...



Linear **C**ongruential **G**enerator (**LCG**):

$X_{i+1} = (A * X_i + C) \bmod M \qquad i = 0, 1, 2, \ldots$
$X_{i+1} = (1 * X_i + 0) \bmod 5 \qquad X_0 = 4$

M = modulus	= 5	($M > 0$)
C = increment	= 0	($C < M$)
A = multiplier	= 1	($A < M$)
X_0 = first value	= 4	

`Random()` in .NET 6

.NET 6 changes to Random

PR #47085, PR #50297 from Stephen Toub, one of key .NET team engineers:

- Changes the algorithm used by `new Random()` to be smaller and faster
- Adds `Random.Shared` thread-safe instance that can be used from any thread

.NET 6 changes to Random

PR #47085, PR #50297 from Stephen Toub, one of key .NET team engineers:

- Changes the algorithm used by `new Random()` to be smaller and faster
- Adds `Random.Shared` thread-safe instance that can be used from any thread
- Adds:
  - `long NextInt64()` // [0 … long.MaxValue)
  - `long NextInt64(max)` // [0 … max)
  - `long NextInt64(min, max)` // [min … max)
  - `float NextSingle()` // [0.0f … 1.0f)

.NET 6 blog: changes to Random

https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-6/

Stephen Toub, Partner Software Engineer on .NET team:

"...over the years we've been hesitant to change `Random`'s implementation for fear of changing the numerical sequence yielded if someone provided a fixed seed to `Random`'s constructor (which is common); now in .NET 6, just as for derived types, we fall back to the old implementation if a seed is supplied, otherwise preferring the new algorithm. This sets us up for the future where we can freely change and evolve the algorithm used by `new Random()` as better approaches appear."

.NET 6 blog: changes to Random

https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-6/

Stephen Toub, Partner Software Engineer on .NET team:

“Until .NET 6, Random employed the same algorithm it had been using for the last 2 decades, a variant of Knuth’s alg that dates back to the 1980s. That served .NET well, but it was time for an upgrade. A myriad number of pseudo-random algorithms have emerged, and for .NET 6 we picked the xoshiro** family, using xoshiro128** on 32-bit and xoshiro256** on 64-bit. These algorithms were introduced by Blackman and Vigna in 2018, are **very fast**, and yield **good enough** pseudo-randomness for Random‘s needs.
For cryptographically-secure rng, SSC.RNG should be used instead.”

.NET 6 Github: changes to Random

Dan Moseley, Group Manager for .NET libraries team, on xoshiro256**:

“...it looks like we didn't really do a "survey" of the PRNG options…
Since we aren't implementing a seeded API here, which we can always change later, perhaps that doesn't matter very much so long as we're reasonably confident that it's not "worse" since folks seem to be mostly wanting more performance and asking for more randomness.”

“xoshiro256** implementation is super simple so unless we discover a significant flaw I think we should go ahead and use it and lock in the **sweet perf gain**. Then we can always change later.”

What is xoshiroNNN** family of algorithms?

- NNN-bit **s**tate: ulong[] **s**; // 4 ulong values for 256bit; 4 uint values for 128bit

What is xoshiroNNN** family of algorithms?

- NNN-bit **s**tate: ulong[] **s**; // 4 ulong values for 256bit; 4 uint values for 128bit
- xoshiro = xo/shi/ro = XOR (^), SHIFT (<<), ROTATE

What is xoshiroNNN** family of algorithms?

- NNN-bit **s**tate: ulong[] **s**; // 4 ulong values for 256bit; 4 uint values for 128bit
- xoshiro = xo/shi/ro = XOR (^), SHIFT (<<), ROTATE
- ROTATE (left) is: ulong Rotl(ulong v, int c) => (v << c) | (v >> (64 – c));

What is xoshiroNNN** family of algorithms?

- NNN-bit **s**tate: ulong[] **s**; // 4 ulong values for 256bit; 4 uint values for 128bit
- xoshiro = xo/shi/ro = XOR (^), SHIFT (<<), ROTATE
- ROTATE (left) is: ulong Rotl(ulong v, int c) => (v << c) | (v >> (64 – c));
- xoshiro is made of 2 parts: a Linear Engine (**LE**), and a Scrambler (**SC**)
  - **LE**: cycles through internal state **s**
  - **SC**: modifies **LE** output before returning it (pure fn, does not alter state)
    - tries to make output less statistically biased

What is xoshiroNNN** family of algorithms?

- NNN-bit **s**tate: ulong[] **s**; // 4 ulong values for 256bit; 4 uint values for 128bit
- xoshiro = xo/shi/ro = XOR (^), SHIFT (<<), ROTATE
- ROTATE (left) is: ulong Rotl(ulong v, int c) => (v << c) | (v >> (64 – c));
- xoshiro is made of 2 parts: a Linear Engine (**LE**), and a Scrambler (**SC**)
  - **LE**: cycles through internal state **s**
  - **SC**: modifies **LE** output before returning it (pure fn, does not alter state)
    - tries to make output less statistically biased
- 3 xoshiro Scramblers are defined: +, ++, and **
  - ulong Scrambler_Plus(ulong[] **s**) => **s**[0] + **s**[3];
  - ulong Scrambler_PlusPlus(ulong[] **s**) => Rotl(**s**[0] + **s**[3], 7) + **s**[0];
  - ulong Scrambler_StarStar(ulong[] **s**) => Rotl(**s**[1] * 5, 7) * 9;

What is xoshiro Linear Engine (LE)?

```
void Xoshiro_LE(ulong[] s)        // s is 4-value state; xoshiro256 is shown here
{
        ulong temp = s[1] << 17;  // “17” is a magic constant A
        s[2] ^= s[0];
        s[3] ^= s[1];
        s[1] ^= s[2];
        s[0] ^= s[3];
        s[2] ^= temp;
        s[3] = Rotl(s[3], 45);    // “45” is a magic constant B
}
```

What is xoshiro256** algorithm? LE + Scrambler

// Step-0a: state **s** is set to a **random 256-bit seed value** (4 random ulong's)
// Step-0b: if **s** is all-zeroes, GOTO Step-0a (highly unlikely)

What is xoshiro256** algorithm? LE + Scrambler

```
// Step-0a: state s is set to a random 256-bit seed value (4 random ulong's)
// Step-0b: if s is all-zeroes, GOTO Step-0a (highly unlikely)

ulong Xoshiro_StarStar(ulong[] s)
{
    ulong next = Scrambler_StarStar(s); // pure fn of current state
    Xoshiro_LE(s); // cycles state
    return next;
}
```

Zeroland

// Step-0a: state **s** is set to a **random 256-bit seed value** (4 random ulong's)
// Step-0b: if **s** is all-zeroes, GOTO Step-0a (highly unlikely)

So as long as state is not all-zeros, xoshiro always works, right? **No!** 😧

Zeroland

// Step-0a: state **s** is set to a **random 256-bit seed value** (4 random ulong's)
// Step-0b: if **s** is all-zeroes, GOTO Step-0a (highly unlikely)

So as long as state is not all-zeros, `xoshiro` always works, right? **No!** 😧

Zeroland:

- Any state with only a few 1-bits or 0-bits (includes all-0 and all-1 states)
- Linear generators like `xoshiro` take a long time to escape **Zeroland**
  - ie. start generating equi-likely 0/1 bits again

Zeroland

// Step-0a: state **s** is set to a **random 256-bit seed value** (4 random ulong's)
// Step-0b: if **s** is all-zeroes, GOTO Step-0a (highly unlikely)

So as long as state is not all-zeros, xoshiro always works, right? **No!** 😦

Zeroland:

- Any state with only a few 1-bits or 0-bits (includes all-0 and all-1 states)
- Linear generators like xoshiro take a long time to escape **Zeroland**
  - ie. start generating equi-likely 0/1 bits again
- Xoshiro escapes **Zeroland** faster than most, but still suffers from it
- Idea #1: randomize state to make **Zeroland** highly unlikely (ex. 256-bit state)
- Idea #2: cycle state a few times after seeding (ex. 16 times) prior to use

Cryptographically Secure rng (csrng)

xoshiro requires randomness to generate randomness: where does it get it?

- System.Security.Cryptography.RandomNumberGenerator.Fill(ulong[] **s**)

Recall this advice?

"For cryptographically-secure rng, SSC.RNG should be used instead."

Cryptographically Secure rng (csrng)

xoshiro requires randomness to generate randomness: where does it get it?

- System.Security.Cryptography.RandomNumberGenerator.Fill(ulong[] **s**)

Recall this advice?

> ”For cryptographically-secure rng, SSC.RNG should be used instead.”

What is “cryptographically-secure rng”?

1. Produces statistically-valid random output (ie. passes randomness tests)
2. Full knowledge of the algorithm (even xkcd passes this one)
3.
4.

Cryptographically Secure rng (csrng)

xoshiro requires randomness to generate randomness: where does it get it?

- System.Security.Cryptography.RandomNumberGenerator.Fill(ulong[] **s**)

Recall this advice?

> ”For cryptographically-secure rng, SSC.RNG should be used instead.”

What is “cryptographically-secure rng”?

1. Produces statistically-valid random output (ie. passes randomness tests)
2. Full knowledge of the algorithm (even xkcd passes this one)
3. Cannot predict future output from any/all prior outputs (without current state)
4.

Cryptographically Secure rng (csrng)

xoshiro requires randomness to generate randomness: where does it get it?

- System.Security.Cryptography.RandomNumberGenerator.Fill(ulong[] **s**)

Recall this advice?

> ”For cryptographically-secure rng, SSC.RNG should be used instead.”

What is “cryptographically-secure rng”?

1. Produces statistically-valid random output (ie. passes randomness tests)
2. Full knowledge of the algorithm (even xkcd passes this one)
3. Cannot predict future output from any/all prior outputs (without current state)
4. Cannot recover past output from current state (future output will be known)

xoshiro256** is predictable (like the old 1980 Random)

xoshiro is trivially predictable: given only 4 xoshiro outputs, we can:

- Derive all prior outputs
- Predict all future outputs

xoshiro256** is predictable (like the old 1980 Random)

xoshiro is trivially predictable: given only 4 xoshiro outputs, we can:

- Derive all prior outputs
- Predict all future outputs

Any flavor of System.Random() is predictable and not cryptographically secure.
Is predictable Random() a problem? **Yes.** Even if you don't do "cryptography"? Yes.

xoshiro256** is predictable (like the old 1980 Random)

xoshiro is trivially predictable: given only 4 xoshiro outputs, we can:

- Derive all prior outputs
- Predict all future outputs

Any flavor of System.Random() is predictable and not cryptographically secure.
Is predictable Random() a problem? **Yes.** Even if you don't do "cryptography"? Yes.

Are any of these "cryptography"?

- Randomized hashing, sorting, shuffling, sampling, k^{th}-ordered, primality tests

Ex. you use Random() to pick a pivot for a random-pivot quicksort

- O(n*log(n)) expected, but $O(n^2)$ if attacker can predict the pivot & affect input

xoshiro256** is predictable (like the old 1980 Random)

xoshiro is trivially predictable: given only 4 xoshiro outputs, we can:

- Derive all prior outputs
- Predict all future outputs

Any flavor of System.Random() is predictable and not cryptographically secure.
Is predictable Random() a problem? **Yes.** Even if you don't do "cryptography"? Yes.

Are any of these "cryptography"?

- Randomized hashing, sorting, shuffling, sampling, k^{th}-ordered, primality tests

Ex. you use Random() to pick a pivot for a random-pivot quicksort

- O(n*log(n)) expected, but $O(n^2)$ if attacker can predict the pivot & affect input

It's not about "cryptography" – it's about **security** and **correctness**

- Any randomized algorithm with external inputs is potentially vulnerable

Superior randomness vs Inferior randomness

xoshiro-based Random() initializes from RandomNumberGenerator (RNG)

- RNG provides **superior randomness**, but is only used to seed xoshiro
- Why not use RNG for **all** randomness? Ie. why even bother with xoshiro?

Superior randomness vs Inferior randomness

xoshiro-based Random() initializes from RandomNumberGenerator (RNG)

- RNG provides **superior randomness**, but is only used to seed xoshiro
- Why not use RNG for **all** randomness? Ie. why even bother with xoshiro?

#1 reason not to use csrng: inferior performance (inferior to what? xkcd?)
...but what if performance is **fast enough**?

#2

#3

Superior randomness vs Inferior randomness

xoshiro-based Random() initializes from RandomNumberGenerator (RNG)

- RNG provides **superior randomness**, but is only used to seed xoshiro
- Why not use RNG for **all** randomness? Ie. why even bother with xoshiro?

#1 reason not to use csrng: inferior performance (inferior to what? xkcd?)
...but what if performance is **fast enough**?

#2 reason not to use csrng: it does not have Random API
...but that can be easily fixed – we're free to subclass & override Random API

#3

Superior randomness vs Inferior randomness

xoshiro-based Random() initializes from RandomNumberGenerator (RNG)

- RNG provides **superior randomness**, but is only used to seed xoshiro
- Why not use RNG for **all** randomness? Ie. why even bother with xoshiro?

#1 reason not to use csrng: inferior performance (inferior to what? xkcd?)
...but what if performance is **fast enough**?

#2 reason not to use csrng: it does not have Random API
...but that can be easily fixed – we're free to subclass & override Random API

#3 reason not to use csrng:
...there is no reason #3

Superior randomness vs Inferior randomness

xoshiro-based Random() initializes from RandomNumberGenerator (RNG)

- RNG provides **superior randomness**, but is only used to seed xoshiro
- Why not use RNG for **all** randomness? Ie. why even bother with xoshiro?

#1 reason not to use csrng: inferior performance (inferior to what? xkcd?)
...but what if performance is **fast enough**?

#2 reason not to use csrng: it does not have Random API
...but that can be easily fixed – we're free to subclass & override Random API

#3 reason not to use csrng:
...there is no reason #3

Perhaps we could fix #1 and #2

1980~2021: what else has changed?

- Cryptographic primitives are HW-accelerated in most CPUs
  - Superior randomness is cheap in 2021
- Most CPU chips have multiple Cores (multiple CPUs) → scale & throughput

1980~2021: what else has changed?

- Cryptographic primitives are HW-accelerated in most CPUs
  - Superior randomness is cheap in 2021
- Most CPU chips have multiple Cores (multiple CPUs) → scale & throughput
- State-buffer size is rarely a concern
  - Smartphones & PCs count memory in GiB
  - Servers count memory in GiB and TiB
  - OS threads come with 1~2 MiB of stack

1980~2021: what else has changed?

- Cryptographic primitives are HW-accelerated in most CPUs
  - Superior randomness is cheap in 2021
- Most CPU chips have multiple Cores (multiple CPUs) → scale & throughput
- State-buffer size is rarely a concern
  - Smartphones & PCs count memory in GiB
  - Servers count memory in GiB and TiB
  - OS threads come with 1~2 MiB of stack

`Random` and `RNG` are already on a collision/merger course:

- `Random` got thread-safe `.Shared` per-thread state, and now seeds from `RNG`
- `RNG` got `.GetInt32(max)` and `.GetInt32(min, max)` methods from `Random`

1980~2021: what else has changed?

- Cryptographic primitives are HW-accelerated in most CPUs
  - Superior randomness is cheap in 2021
- Most CPU chips have multiple Cores (multiple CPUs) → scale & throughput
- State-buffer size is rarely a concern
  - Smartphones & PCs count memory in GiB
  - Servers count memory in GiB and TiB
  - OS threads come with 1~2 MiB of stack

`Random` and RNG are already on a collision/merger course:

- `Random` got thread-safe `.Shared` per-thread state, and now seeds from RNG
- RNG got `.GetInt32(max)` and `.GetInt32(min, max)` methods from `Random`

Can we make a randomness-provider wishlist 2021?

Randomness provider wishlist 2021

1. **Drop-in replacement for Random** (subclasses System.Random)

Randomness provider wishlist 2021

1. **Drop-in replacement for Random** (subclasses System.Random)
2. **Cryptographically-strong randomness only** (superior-quality randomness)
    - meets all cs-rng criteria (backtrack resistant, state-leak resistant, etc.)

Randomness provider wishlist 2021

1. **Drop-in replacement for Random** (subclasses System.Random)
2. **Cryptographically-strong randomness only** (superior-quality randomness)
   - meets all cs-rng criteria (backtrack resistant, state-leak resistant, etc.)
3. **100% Thread-safe** (every API)

Randomness provider wishlist 2021

1. **Drop-in replacement for Random** (subclasses System.Random)
2. **Cryptographically-strong randomness only** (superior-quality randomness)
   - meets all cs-rng criteria (backtrack resistant, state-leak resistant, etc.)
3. **100% Thread-safe** (every API)
4. **Fast enough** (as default choice for 99% of all typical randomness needs)
   - perf scales per-CPU/Core (ie. thread-safety is not via 1-lock contention)

Randomness provider wishlist 2021

1. **Drop-in replacement for Random** (subclasses System.Random)
2. **Cryptographically-strong randomness only** (superior-quality randomness)
   - meets all cs-rng criteria (backtrack resistant, state-leak resistant, etc.)
3. **100% Thread-safe** (every API)
4. **Fast enough** (as default choice for 99% of all typical randomness needs)
   - perf scales per-CPU/Core (ie. thread-safety is not via 1-lock contention)
5. **Unseeded and seeded**
   - seeded is guaranteed to be reproducible (on every CPU/platform)
   - seeded state is 256 bits (ex. “shuffle a deck of 52 cards” is 2^{226} shuffles)

Randomness provider wishlist 2021

1. **Drop-in replacement for Random** (subclasses System.Random)
2. **Cryptographically-strong randomness only** (superior-quality randomness)
   - meets all cs-rng criteria (backtrack resistant, state-leak resistant, etc.)
3. **100% Thread-safe** (every API)
4. **Fast enough** (as default choice for 99% of all typical randomness needs)
   - perf scales per-CPU/Core (ie. thread-safety is not via 1-lock contention)
5. **Unseeded and seeded**
   - seeded is guaranteed to be reproducible (on every CPU/platform)
   - seeded state is 256 bits (ex. “shuffle a deck of 52 cards” is 2^{226} shuffles)

Can it be built?
Would it be fast enough?

Randomness provider wishlist 2021

1. **Drop-in replacement for Random** (subclasses System.Random)
2. **Cryptographically-strong randomness only** (superior-quality randomness)
   - meets all cs-rng criteria (backtrack resistant, state-leak resistant, etc.)
3. **100% Thread-safe** (every API)
4. **Fast enough** (as default choice for 99% of all typical randomness needs)
   - perf scales per-CPU/Core (ie. thread-safety is not via 1-lock contention)
5. **Unseeded and seeded**
   - seeded is guaranteed to be reproducible (on every CPU/platform)
   - seeded state is 256 bits (ex. “shuffle a deck of 52 cards” is 2^{226} shuffles)

Can it be built?
Would it be fast enough?

System.Random:
- Convenient API
- High Performance

SSC.RNG:
- Superior Randomness
- Thread Safety

CryptoRandom – modern replacement for Random & RNG

Can it be built?

- CryptoRandom .NET library (Nuget: “CryptoRandom”)
- Implements everything on the wishlist
- Uses tiny per-Core state buffers (<0.8% of 1Mb)

CryptoRandom – modern replacement for Random & RNG

Can it be built?

- CryptoRandom .NET library (Nuget: “CryptoRandom”)
- Implements everything on the wishlist
- Uses tiny per-Core state buffers (<0.8% of 1Mb)

Is it fast enough?

- byte throughput
- call throughput

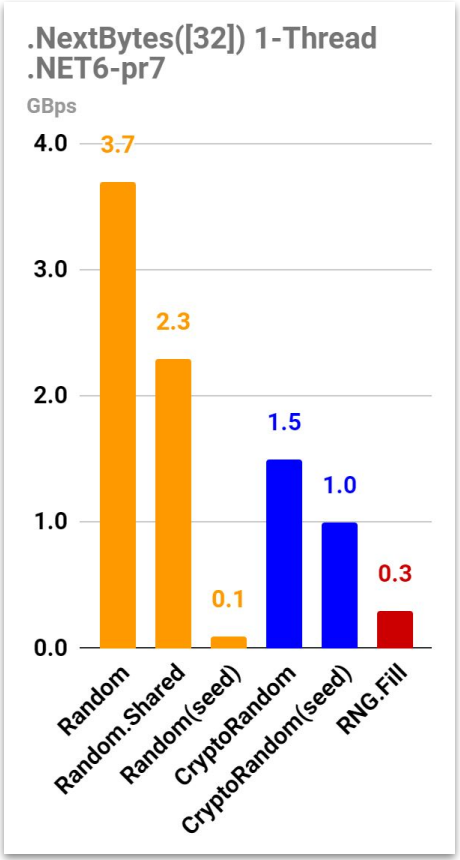
…but what is “fast enough”?

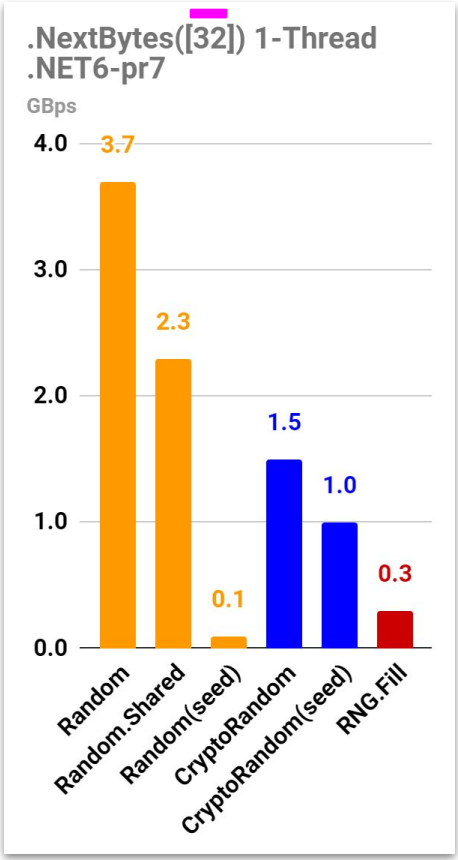
- Fastest Local Devices 2021:
  - Thunderbolt 3: ~5.0 GBps
  - USB 3.2: ~2.0 GBps
  - 10G Ethernet: ~1.2 GBps
  - 25G Ethernet: ~3.1 GBps
  - Fastest SSD drives: ~3.1 GBps (https://ssd.userbenchmark.com/)

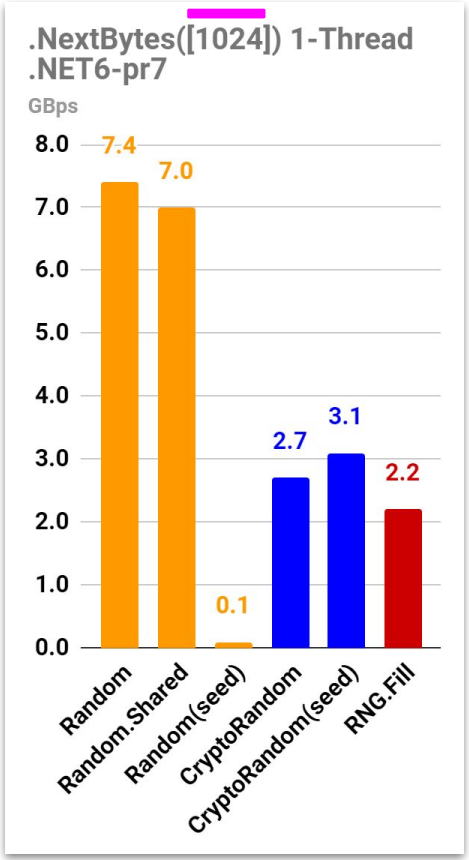
…but what is “fast enough”?

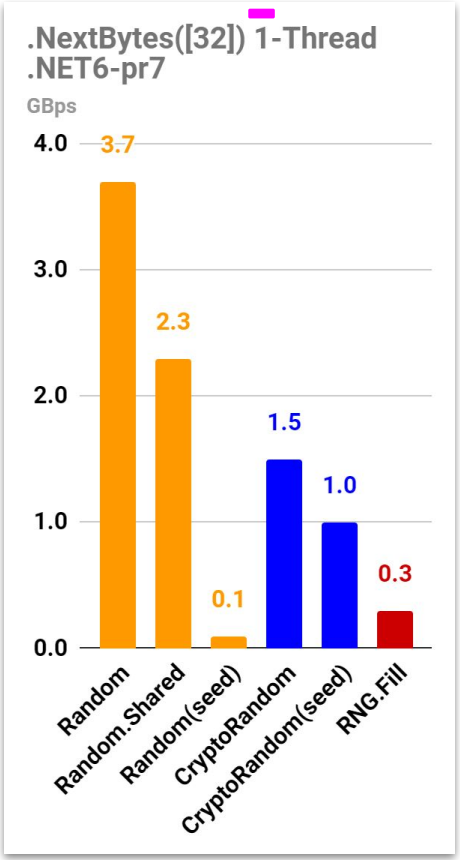
- Fastest Local Devices 2021:
  - Thunderbolt 3: ~5.0 GBps
  - USB 3.2: ~2.0 GBps
  - 10G Ethernet: ~1.2 GBps
  - 25G Ethernet: ~3.1 GBps
  - Fastest SSD drives: ~3.1 GBps (https://ssd.userbenchmark.com/)

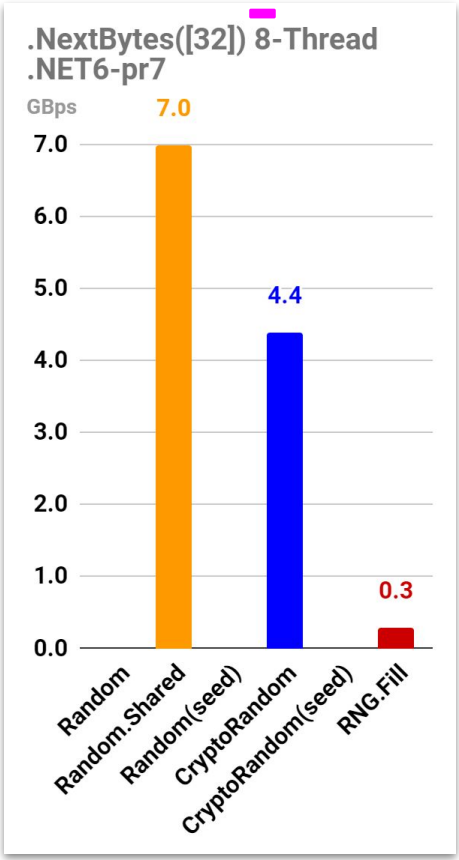
- Fastest durable Cloud Storage 2021:
  - Alibaba ~4.0 GBps
  - Azure: ~2.0 GBps
  - AWS: ~4.0 GBps
  - GCP: ~2.2 GBps

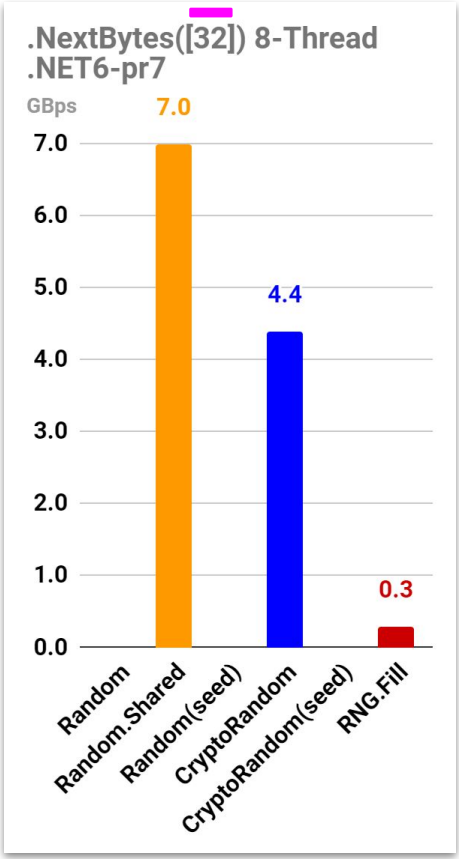

.NextBytes([32]) 1-Thread
.NET6-pr7
GBps
4.0
3.0
2.0
1.0
0.0
3.7
2.3
0.1
1.5
1.0
0.3
Random
Random.Shared
Random(seed)
CryptoRandom
CryptoRandom(seed)
RNG.Fill

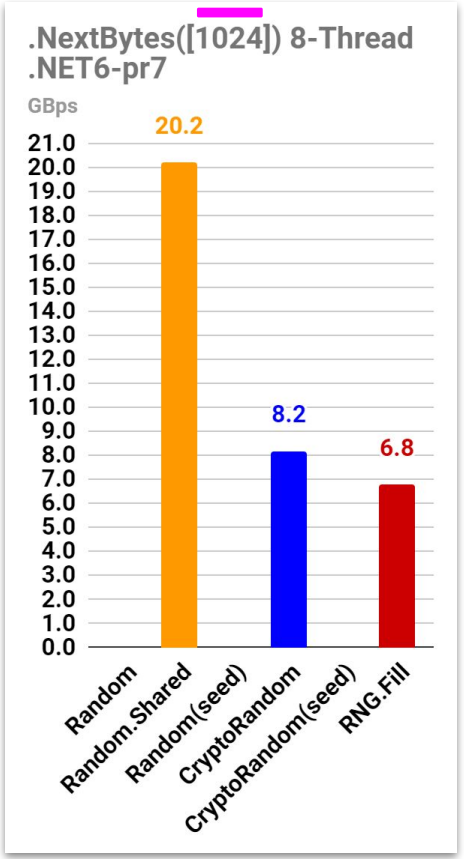

.NextBytes([32]) 1-Thread
.NET6-pr7
GBps
4.0
3.0
2.0
1.0
0.0
3.7
2.3
0.1
1.5
1.0
0.3
Random
Random.Shared
Random(seed)
CryptoRandom
CryptoRandom(seed)
RNG.Fill

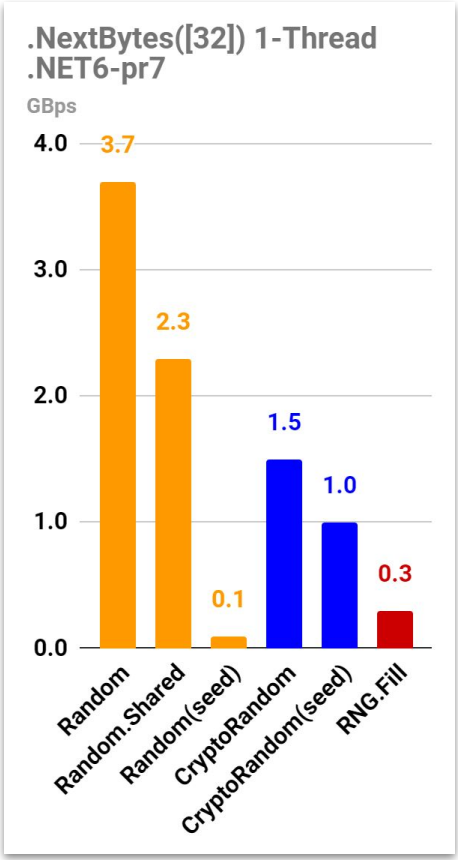

.NextBytes([1024]) 1-Thread
.NET6-pr7
GBps
8.0
7.0
6.0
5.0
4.0
3.0
2.0
1.0
0.0
7.4
7.0
0.1
2.7
3.1
2.2
Random
Random.Shared
Random(seed)
CryptoRandom
CryptoRandom(seed)
RNG.Fill

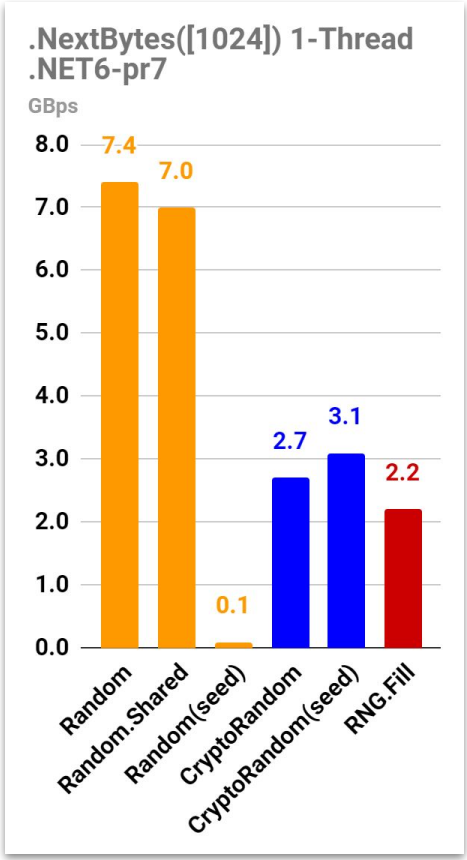

.NextBytes([32]) 1-Thread
.NET6-pr7
GBps
4.0
3.0
2.0
1.0
0.0
3.7
2.3
0.1
1.5
1.0
0.3
Random
Random.Shared
Random(seed)
CryptoRandom
CryptoRandom(seed)
RNG.Fill

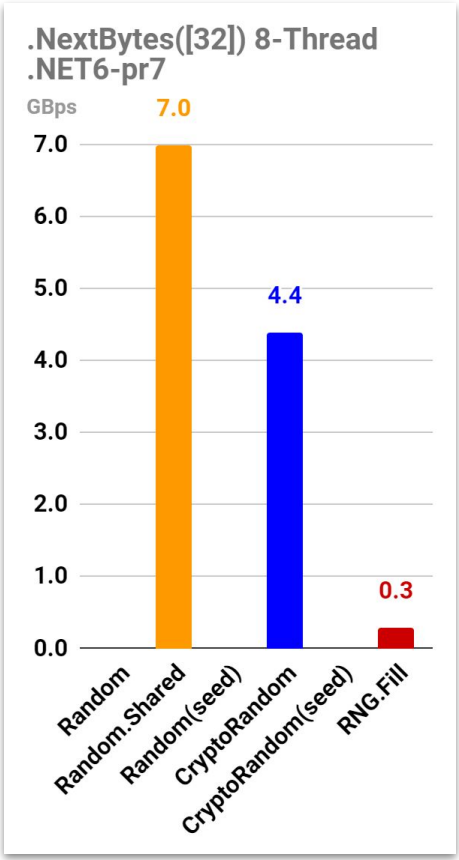

.NextBytes([32]) 8-Thread
.NET6-pr7
GBps
7.0
6.0
5.0
4.0
3.0
2.0
1.0
0.0
7.0
4.4
0.3
Random
Random.Shared
Random(seed)
CryptoRandom
CryptoRandom(seed)
RNG.Fill

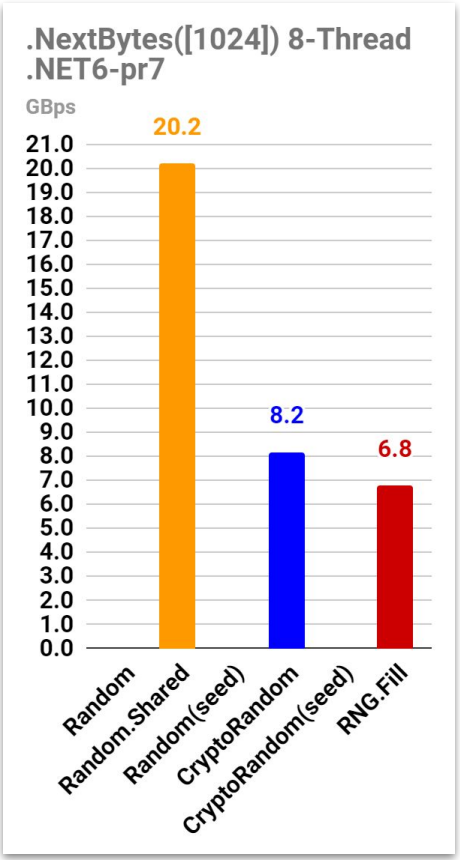

.NextBytes([32]) 8-Thread
.NET6-pr7
GBps
7.0
4.4
0.3
7.0
6.0
5.0
4.0
3.0
2.0
1.0
0.0
Random
Random.Shared
Random(seed)
CryptoRandom
CryptoRandom(seed)
RNG.Fill


.NextBytes([1024]) 8-Thread
.NET6-pr7
GBps
20.2
8.2
6.8
21.0
20.0
19.0
18.0
17.0
16.0
15.0
14.0
13.0
12.0
11.0
10.0
9.0
8.0
7.0
6.0
5.0
4.0
3.0
2.0
1.0
0.0
Random
Random.Shared
Random(seed)
CryptoRandom
CryptoRandom(seed)
RNG.Fill









- Old Random (.NET <=5): ~0.1 GBps (1 thread, old laptop)
- New Random (.NET 6): ~7 GBps (1 thread, old laptop)
- CryptoRandom: ~3 GBps (1 thread, old laptop)

Call throughput of.Next() / .GetInt32()

Multi-threaded call-frequency perf:

- .Next() for Random API, and
- .GetInt32() for RNG

Old Random needs a Lock (.NET<=5).

Random.Shared is .NET 6+ only.

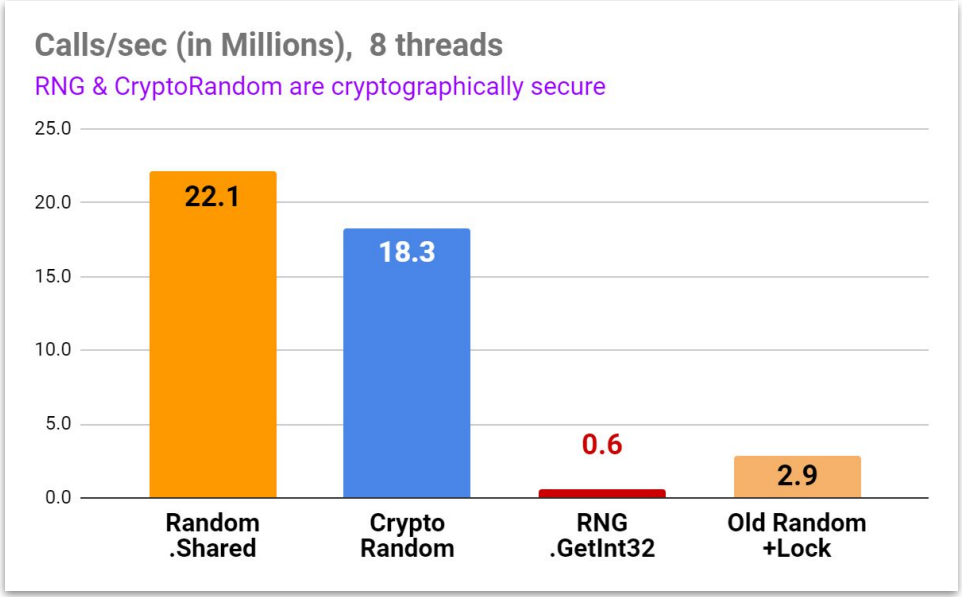
Call throughput of.Next() / .GetInt32()

Multi-threaded call-frequency perf:

- .Next() for Random API, and
- .GetInt32() for RNG

Old Random needs a Lock (.NET<=5).

Random.Shared is .NET 6+ only.



Call throughput of.Next() / .GetInt32()

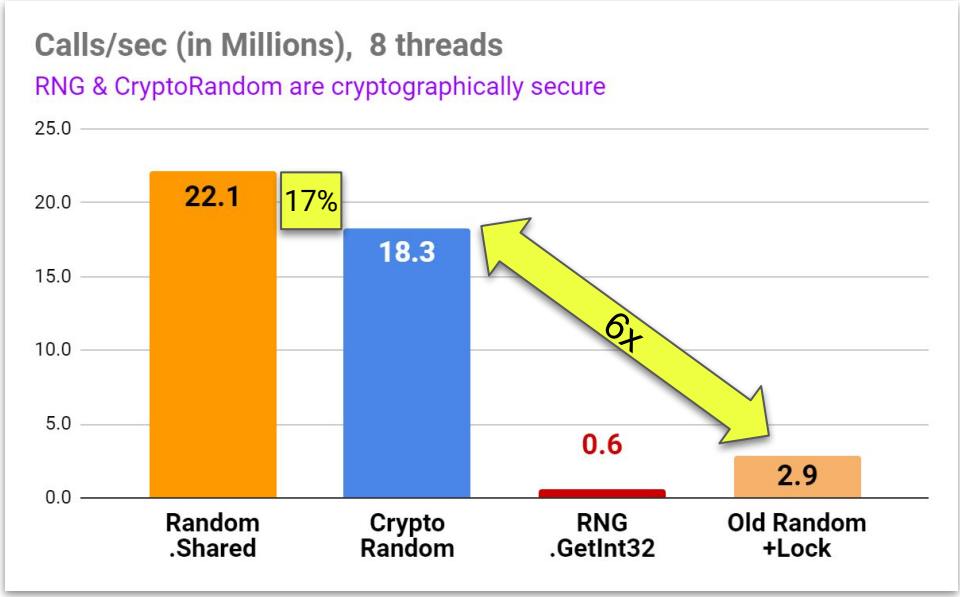
Multi-threaded call-frequency perf:

- .Next() for Random API, and
- .GetInt32() for RNG

Old Random needs a Lock (.NET<=5).

Random.Shared is .NET 6+ only.

Is CryptoRandom fast enough? Yes.



new .NET 6 Random & CryptoRandom

- .NET 6 `Random()`/`xoshiro256**` predictable linear pseudo-random generator:
  - Very fast (`xkcd` random is even faster 😊)
  - Does not suffer from most of the failures of old Random
  - Not seeded, not for .NET <6, not for secure (most?) uses

new .NET 6 Random & CryptoRandom

- .NET 6 `Random()`/`xoshiro256**` predictable linear pseudo-random generator:
  - Very fast (`xkcd` random is even faster 😊)
  - Does not suffer from most of the failures of old Random
  - Not seeded, not for .NET <6, not for secure (most?) uses

- CryptoRandom:
  - Everything on the wishlist – superior secure randomness only
  - Replaces seeded/unseeded `Random()` and `RandomNumberGenerator`

new .NET 6 Random & CryptoRandom

- .NET 6 `Random()`/xoshiro256** predictable linear pseudo-random generator:
  - Very fast (`xkcd` random is even faster 😊)
  - Does not suffer from most of the failures of old Random
  - Not seeded, not for .NET <6, not for secure (most?) uses

- CryptoRandom:
  - Everything on the wishlist – superior secure randomness only
  - Replaces seeded/unseeded `Random()` and `RandomNumberGenerator`
  - `CR.NextGuid()` is ~10x faster vs `Guid.NewGuid()` on Windows
    - ~30x faster on Linux per-Core (unresolved Github issue for 2 years)

new .NET 6 Random & CryptoRandom

- .NET 6 `Random()`/xoshiro256** predictable linear pseudo-random generator:
  - Very fast (`xkcd` random is even faster 😊)
  - Does not suffer from most of the failures of old Random
  - Not seeded, not for .NET <6, not for secure (most?) uses

- CryptoRandom:
  - Everything on the wishlist – superior secure randomness only
  - Replaces seeded/unseeded `Random()` and `RandomNumberGenerator`
  - `CR.NextGuid()` is ~10x faster vs `Guid.NewGuid()` on Windows
    - ~30x faster on Linux per-Core (unresolved Github issue for 2 years)
  - Fast enough for 99% of uses
  - Consider as your **new default** randomness provider

In conclusion

- New `xoshiro`-based `Random()` is much better and faster in .NET 6 ✓

In conclusion

- New `xoshiro`-based `Random()` is much better and faster in .NET 6 ✓
- `Random(Seed)` is the same old sadness, forever
  - Avoid it if you can
  - No seeded `xoshiro` in .NET 6 (for good reasons)

In conclusion

- New `xoshiro`-based `Random()` is much better and faster in .NET 6 ✓
- `Random(Seed)` is the same old sadness, forever
  - Avoid it if you can
  - No seeded `xoshiro` in .NET 6 (for good reasons)
- Stop new'ing `Random()` – use `Random.Shared` instead in .NET 6+
  - Audit code for "new Random", change it

In conclusion

- New `xoshiro`-based `Random()` is much better and faster in .NET 6 ✓
- `Random(Seed)` is the same old sadness, forever
  - Avoid it if you can
  - No seeded `xoshiro` in .NET 6 (for good reasons)
- Stop new'ing `Random()` – use `Random.Shared` instead in .NET 6+
  - Audit code for "new Random", change it
- No excuse not to default to **superior quality randomness** in 2021
  - It's not "*do I need it?*" – you always need it, should always start with it
  - Performance can be good enough, but not Randomness:
    - it should either be cryptographically strong, or "avoid it" kind

In conclusion

- New `xoshiro`-based `Random()` is much better and faster in .NET 6 ✓
- `Random(Seed)` is the same old sadness, forever
  - Avoid it if you can
  - No seeded `xoshiro` in .NET 6 (for good reasons)
- Stop new'ing `Random()` – use `Random.Shared` instead in .NET 6+
  - Audit code for "new Random", change it
- No excuse not to default to **superior quality randomness** in 2021
  - It's not "*do I need it?*" – you always need it, should always start with it
  - Performance can be good enough, but not Randomness:
    - it should either be cryptographically strong, or "avoid it" kind
- **CryptoRandom** is a fast, safe, secure drop-in for `Random`

Links

- Xoshiro generator family – David Blackman and Sebastiano Vigna
  - https://prng.di.unimi.it/
  - https://vigna.di.unimi.it/ftp/papers/ScrambledLinear.pdf
- Exploring Xoshiro Zeroland – Melissa O'Neill
  - https://www.pcg-random.org/posts/xoshiro-repeat-flaws.html
- .NET 6 implementation of Xoshiro
  - https://github.com/dotnet/runtime/blob/main/src/libraries/System.Private.CoreLib/src/System/Random.Xoshiro256StarStarImpl.cs
- DEF CON 29 – Dan Petro – You're doing IoT RNG
  - https://www.youtube.com/watch?v=Zuqw0-jZh9Y

Thank you!

Questions?

sdrapkin@sdprime.com

github.com/sdrapkin

Appendix

Other .NET 6 changes related to randomness

```
var r1 = RandomNumberGenerator.Create();
var r2 = RandomNumberGenerator.Create();
object.ReferenceEquals(r1, r2) ?
```

`False` in .NET <=5

- new RNG implementation object is created on every call

`True` in .NET 6

- RandomNumberGenerator.Create() returns a singleton object
- `Dispose()` is a no-op (calls GC.SuppressFinalize(this))

Other .NET 6 changes related to randomness

- `Guid.NewGuid()` is now guaranteed to contain 122 cryptographically strong random bits, on all .NET platforms (PR 42770)

- FastGuid .NET library (Nuget: “FastGuid”)
  - creates 128-bit strongly-random Guids
  - 10x faster on Windows vs `Guid.NewGuid()`
  - 30x faster on non-Windows vs `Guid.NewGuid()`
  - Scales per-core (ex. 4-core Linux → ~80x faster)

```
[Host] : .NET 6.0.0 (6.0.21.48005), X64 RyuJIT
|           Method |       Mean |    Error |   StdDev | Ratio |
|----------------- |-----------:|---------:|---------:|------:|
| FastGuid_NewGuid |   102.9 ns |  0.97 ns |  0.81 ns |  1.00 |
|     Guid_NewGuid | 1,101.9 ns | 21.76 ns | 30.50 ns | 10.67 |
```