RandomSHAKE

A CSPRNG backed by (Turbo)SHAKE256

ESCADA meeting on Nov 14, 2025

Presentation by Anjan Roy

I’m pursuing PhD in Radboud University, Nijmegen, The Netherlands under Joan Daemen

What is RandomSHAKE?

A cryptographically secure pseudo-random number generator (CSPRNG), backed by (Turbo)SHAKE256 eXtendable Output Function (XOF), with occasional ratcheting, offering 128, 192 and 256-bits of security.

Why do we need RandomSHAKE?

- Need randomness in testing PQC+LWC impl.
- NIST has a DRBG: AES256 in CTR mode.
- But need OpenSSL as a dependency.
- OpenSSL is big and the UX is not best.
- Need something low-footprint, portable and easy-to-use.

NIST Deterministic Random Bit Generator (DRBG)

Powered by AES256 in CTR mode

Code Blame Raw

```c
//
//  rng.h
//
//  Created by Bassham, Lawrence E (Fed) on 8/29/17.
//  Copyright © 2017 Bassham, Lawrence E (Fed). All rights reserved.
//

#ifndef rng_h
#define rng_h

#include <stdio.h>

#define RNG_SUCCESS      0
#define RNG_BAD_MAXLEN  -1
#define RNG_BAD_OUTBUF  -2
#define RNG_BAD_REQ_LEN -3

typedef struct {
    unsigned char   buffer[16];
    int             buffer_pos;
    unsigned long   length_remaining;
    unsigned char   key[32];
    unsigned char   ctr[16];
} AES_XOF_struct;

typedef struct {
    unsigned char   Key[32];
    unsigned char   V[16];
    int             reseed_counter;
} AES256_CTR_DRBG_struct;


void
AES256_CTR_DRBG_Update(unsigned char *provided_data,
                       unsigned char *Key,
                       unsigned char *V);

int
seedexpander_init(AES_XOF_struct *ctx,
                  unsigned char *seed,
                  unsigned char *diversifier,
                  unsigned long maxlen);

int
seedexpander(AES_XOF_struct *ctx, unsigned char *x, unsigned long xlen);

void
randombytes_init(unsigned char *entropy_input,
                 unsigned char *personalization_string,
                 int security_strength);

int
randombytes(unsigned char *x, unsigned long long xlen);

#endif /* rng_h */
```

Enter RandomSHAKE

- It’s backed by SHAKE256 or TurboSHAKE256 XOF.
- It’s keccak permutation based.
- It’s sponge mode of operation based.
- It’s implemented as header-only portable C++ library.
- Uses modern C++ features for better UX.
- Integrates nicely with C++ <*random*> header in stdlib.

Why not use (Turbo)SHAKE128?

- RandomSHAKE offers three levels of bit security i.e. 128, 192 and 256-bits.
- Your choice of bit-security level dictates how long seed RandomSHAKE expects from you.
- Because (Turbo)SHAKE256 offers at max 256-bits of security, seems safe to base the CSPRNG on it instead of *fragmenting* API with (Turbo)SHAKE128.

What is the deal with ratcheting?

```
53  // Specialization for SHAKE256 XOF.
    You, 12 hours ago | 1 author (You)
54  template<>
55  struct xof_selector_t<xof_kind_t::SHAKE256>
56  {
57    using type = shake256::shake256_t;
58    static constexpr size_t rate = shake256::RATE;
59
60    // Everytime these many bytes are squeezed from underlying sponge, we zeroize first `n` bytes of Keccak permutation
61    // state and re-apply 24-rounds permutation.
62    static constexpr size_t ratchet_period_byte_len = shake256::RATE;
63  };      You, 12 hours ago • Add support for using TurboSHAKE256 XOF (the de…
64
```

Produce 1088 pseudo-random bytes from CSPRNG
then zeroize first **n**-bytes of keccak permutation state
and permute the state with 12 or 24-rounds keccak. **Repeat.**

How many bytes to zeroize during ratcheting?

```
90  struct randomshake_t
91  {
92  private:
93    xof_selector_t<xof_kind>::type state{};
94    std::array<uint8_t, xof_selector_t<xof_kind>::ratchet_period_byte_len> buffer{};
95    size_t buffer_offset = 0u;
96
97    // These many bytes are zeroed from the beginning of the Keccak permutation state during ratchet operation.
98    static constexpr size_t ratchet_byte_len = std::min(xof_selector_t<xof_kind>::rate, bit_security_level) / 8u;      You, 13 hours ag
99
100 public:
101   using result_type = UIntType;
102
```

Rate = 1088b for (Turbo)SHAKE256

Depends mostly on bit-security level of the CSPRNG

- 128b CSPRNG zeroizes first 16B state
- 192b CSPRNG zeroizes first 24B state
- 256b CSPRNG zeroizes first 32B state

How to actually ratchet?

```
100
101  /**      You, 4 days ago • Introduce oneshot hasher API for SHA3 hash func…
102   * Given that sponge is already finalized, this routine can be used for zeroizing first n -bytes of
103   * permutation state s.t. n <= 200 and applying permutation.
104   */
105  forceinline void ratchet(const size_t byte_len)
106  {
107    if (finalized) {
108      const auto ratchetable_portion_byte_len = std::min(byte_len, keccak::STATE_BYTE_LEN);
109
110      auto state_as_bytes = reinterpret_cast<uint8_t*>(state);
111      std::memset(state_as_bytes, 0, ratchetable_portion_byte_len);
112
113      keccak::permute<NUM_KECCAK_ROUNDS>(state);
114    }
115  }
```

1) Zeroize

2) Permute

RandomSHAKE in C++

- Single-file C++ header-only library impl.
- Dependency on *sha3* header-only C++ lib.
- *sha3* is ~10-15% slower compared to XKCP.
- Well integrated with C++ <*random*> header.

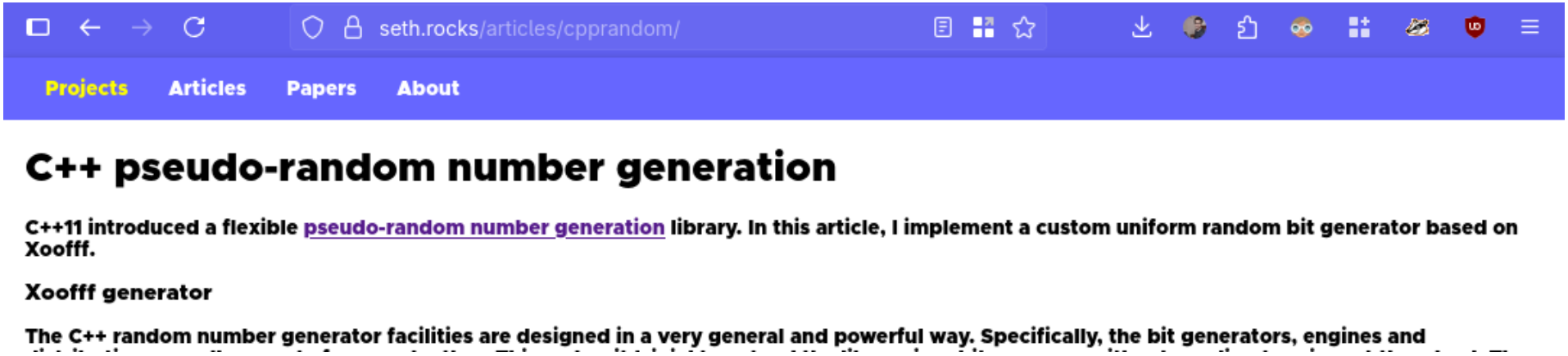


C++ <random> header = random bit generation engine + statistical distribution

Modular Design – Non-intrusive, zero-boilerplate

sha3: https://github.com/itzmeanjan/sha3

Inspiration behind RandomSHAKE



RandomSHAKE is inspired by Seth Hoffert's blog post titled
"**C++ pseudo-random number generation**"

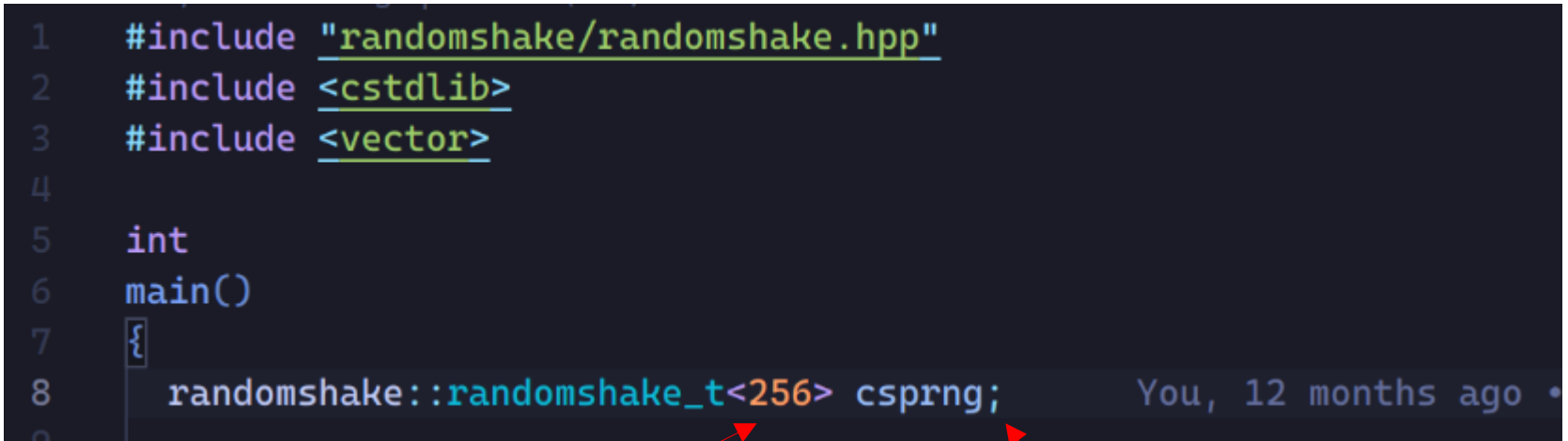
Backed by Xoofff = Farfalle+Xoodoo
Parallel Permutation-based Crypto = GB/s throughput with AVX512

Read full blog @ https://seth.rocks/articles/cpprandom/

How to use RandomSHAKE?

- Init i.e. actually seed (Turbo)SHAKE256 XOF.
- Generate arbitrary long random bytes or unsigned integers (u8, u16, u32 and u64).
- Plug with any C++ stdlib distributions such as uniform, poisson, normal, bernoulli etc. from <*random*>.
- C++ <*random*> header introduced in C++11.

How to init RandomSHAKE CSPRNG?



Sample 256-bits of entropy from *std::random_device*
Non-deterministic random number generator, sampling from h/w sources

Convenient though *potentially* unsafe due to implementation-defined behaviour

How to init RandomSHAKE CSPRNG?

```cpp
#include "randomshake/randomshake.hpp"
#include <array>
#include <cstdint>
#include <cstdlib>
#include <random>
#include <ranges>

int
main()
{
  std::array<uint8_t, randomshake::randomshake_t<256>::seed_byte_len> seed{};
  seed.fill(0xfe);

  // Deterministic CSPRNG : Seed -based initialization of CSPRNG
  randomshake::randomshake_t<256> csprng(seed);
```

Seeded init, deterministic, much safer, though less convenient

How to generate random bits using RandomSHAKE?

Line-by-line code walkthrough 🚀

```cpp
#include "randomshake/randomshake.hpp"
#include <cstdint>
#include <cstdlib>

int
main()
{
  // 192-bit security CSPRNG, producing uint8_t values.
  randomshake::randomshake_t<192> csprng_u8;

  // Sample a single random uint8_t.
  const auto random_u8 = csprng_u8();

  // or
  // Fill the vector, sampling one byte at a time.
  std::vector<uint8_t> rand_u8_values(16, 0x00);
  std::ranges::generate(rand_u8_values, [&]() { return csprng_u8(); });

  // or
  // Fill the vector, by squeezing arbitrary many bytes at a time.
  csprng_u8.generate(rand_u8_values);

  // ------------------------------------------------------------------------

  // 192-bit security CSPRNG, producing uint32_t values.
  randomshake::randomshake_t<192, uint32_t> csprng_u32;

  // Sample a single random uint32_t.
  const auto random_u32 = csprng_u32();

  // or
  // Fill the vector, sampling one u32 at a time.
  std::vector<uint32_t> rand_u32_values(4, 0x00);
  std::ranges::generate(rand_u32_values, [&]() { return csprng_u32(); });

  // Should fail to fill the vector, with CSPRNG of wrong type.
  csprng_u32.generate(rand_u32_values);

  // Should fail to instantiate CSPRNG with unsupported bit-security level.
  randomshake::randomshake_t<191> csprng_192b;

  return EXIT_SUCCESS;
}
```

How to plug C++ stdlib distribution?

Number of HEADs in 1000 coin-flips

```cpp
randomshake::randomshake_t<256> csprng;
std::binomial_distribution dist{ 1'000, .5 };

for (auto _ : std::ranges::iota_view{ 1, 10 }) {
  std::cout << "[BINOMIAL dISTRIBUTION] Number of heads in 1,000 flips: " << dist(csprng) << '\n';
}
```

Generate random number as per distribution

```
--- --- ---
[BINOMIAL dISTRIBUTION] Number of heads in 1,000 flips: 490
[BINOMIAL dISTRIBUTION] Number of heads in 1,000 flips: 493
[BINOMIAL dISTRIBUTION] Number of heads in 1,000 flips: 488
[BINOMIAL dISTRIBUTION] Number of heads in 1,000 flips: 498
[BINOMIAL dISTRIBUTION] Number of heads in 1,000 flips: 505
[BINOMIAL dISTRIBUTION] Number of heads in 1,000 flips: 510
[BINOMIAL dISTRIBUTION] Number of heads in 1,000 flips: 493
[BINOMIAL dISTRIBUTION] Number of heads in 1,000 flips: 511
[BINOMIAL dISTRIBUTION] Number of heads in 1,000 flips: 502
--- --- ---
```

Result of 1000 coin-flips

How to generate uniform rand. ints?

```cpp
randomshake::randomshake_t<128> csprng;
std::uniform_int_distribution<uint8_t> dist{ 97, 102 };

std::cout << "10 random values in range [a, f]: ";
for (auto _ : std::ranges::iota_view{ 0, 10 }) {
  std::cout << dist(csprng);
}
std::cout << "\n";

// ---------------------------------------------------------------------        You, 1 second ago • Uncommitted changes

std::uniform_int_distribution<uint64_t> dist_large{ UINT64_MAX / 2, UINT64_MAX };

std::cout << "5 random u64 values: ";
for (auto _ : std::ranges::iota_view{ 0, 5 }) {
  std::cout << dist_large(csprng) << ", ";
}
std::cout << "\n";

return EXIT_SUCCESS;
```

(1)

(2)

Init the CSPRNG once.
Then use it with different distributions.

Console output

```
--- --- ---
10 random values in range [a, f]: faddbfbced
5 random u64 values: 17288455242315046306, 15319303753528212563, 13747154517105190476, 13304461696698666857, 11933463920859584881,
--- --- ---
```

Can I generate uniform random floats?

Yes.

- Generate 1M random floats.
- Check if they are uniformly distributed.
- Use formula vs. over the data.

```
--- --- ---
Expected standard deviation = 0.288675
Computed standard deviation = 0.288874
Difference                  = 0.000198573
--- --- ---
```

Works !

```cpp
int
main()
{
  randomshake::randomshake_t<128> csprng;
  std::uniform_real_distribution<float> dist{ 0.0, 1.0 };

  std::vector<float> rand_vals(1'000'000, 0.f);
  std::ranges::generate(rand_vals, [&]() { return dist(csprng); });

  const auto expected = expected_continuous_uniform_distribution_sd(0.f, 1.f);
  const auto computed = compute_sd(rand_vals);

  std::cout << "Expected standard deviation = " << expected << "\n";
  std::cout << "Computed standard deviation = " << computed << "\n";
  std::cout << "Difference                  = " << std::abs(expected - computed) << "\n";

  return EXIT_SUCCESS;
}

float
compute_mean(std::span<float> vals)
{
  float sum = 0;
  for (auto v : vals) {
    sum += v;
  }

  return sum / static_cast<float>(vals.size());
}

float
compute_sd(std::span<float> vals)
{
                                                    2.f);

                                                iff_sum / static_cast<float>(vals.size());

float
expected_continuous_uniform_distribution_sd(const float lo, const float hi)
{
  return (hi - lo) / std::sqrt(12.f);
}
```

How fast is RandomSHAKE?

Benchmark	Time	CPU	Iterations	CYCLES	bytes_per_second
csprng/shake256/192b/generate_byte_seq_mean	1827699 ns	1827274 ns	10	7.4742M	547.293Mi/s
csprng/shake256/192b/generate_byte_seq_median	1825112 ns	1824753 ns	10	7.47804M	548.021Mi/s
csprng/shake256/192b/generate_byte_seq_stddev	14102 ns	14105 ns	10	17.6488k	4.22208Mi/s
csprng/shake256/192b/generate_byte_seq_cv	0.77 %	0.77 %	10	0.24%	0.77%
csprng/shake256/192b/generate_byte_seq_min	1805758 ns	1805205 ns	10	7.44802M	540.939Mi/s
csprng/shake256/192b/generate_byte_seq_max	1849012 ns	1848638 ns	10	7.49231M	553.954Mi/s
csprng/turboshake256/192b/generate_byte_seq_mean	978906 ns	978674 ns	10	4.16992M	0.999957Gi/s
csprng/turboshake256/192b/generate_byte_seq_median	980664 ns	980430 ns	10	4.17159M	1020.74Mi/s
csprng/turboshake256/192b/generate_byte_seq_stddev	47166 ns	47147 ns	10	10.7193k	49.9687Mi/s
csprng/turboshake256/192b/generate_byte_seq_cv	4.82 %	4.82 %	10	0.26%	4.88%
csprng/turboshake256/192b/generate_byte_seq_min	900496 ns	900307 ns	10	4.14712M	969.259Mi/s
csprng/turboshake256/192b/generate_byte_seq_max	1031934 ns	1031716 ns	10	4.18632M	1.0847Gi/s

On Intel x86_64, running Linux kernel 6.17.0
Compiled with GCC 15.2.0 and all optimizations turned on "*-O3 -march=native -flto*"

Like RandomSHAKE?

Head over to
https://github.com/itzmeanjan/RandomShake

Read the README.md
And C++ <random> header documentation
https://en.cppreference.com/w/cpp/numeric/random.html
for exploring full potential.

RandomSHAKE

Thanks for your time and attention.

