The implementation of *crush*

Daniel Chicayban Bastos
Luis Antonio Brasil Kowada

May 5th 2019

This document

This document describes the implementation of *crush*, a program for testing random number generators using the TestU01 library [5, 4, 6]. It is at the same time the program's source code and its documentation: it is written in the literate programming [2, 3, 7] spirit.

A literate program interleaves program source code and documentation in the same document. When chunks of code are displayed on a page of this document, you will find an integer on the left margin of that page. This integer specifies the page on which the chunk of code is written.

If a chunk is referenced by an integer and by a letter, that means there's more than one chunk defined on the same page, so letters distinguish chunks defined on the same page.

The *crush* program

The program implemented in this document is called *crush*.

```
%./crush --help
Usage: ./crush [options]
 Tests your data for randomness against TestU01.

Examples:
cat /dev/urandom | crush -b small -n 'the local generator'
xorshift32 | crush -b small -n xorshift32

 The options are:
-b, --battery    Your choice of battery (small, medium, big)
-n, --name       The name of your generator
-h, --help       Display this information
%
```

Suppose we have a program called *xs32*, which implements George Marsaglia's *xorshift* pseudo-random number generator with an output size of 32 bits. Assume *xs32* writes its random numbers in binary, in little-endian format, to the standard output. To test this generator against the small battery of the library TestU01, we can say the following to the UNIX shell:

```
%./xs32 | crush -b small -n xs32
[...]
%
```

Where we wrote "[...]", we would see a report from the library describing the results of each statistical test applied to the generator.

The option *-n* sets the name of the generator, a mere formality of the TestU01 library. As the library produces the report, it conveniently annotates it with the generator's name for our later reference.

It the next sections of this document, we write the code that implements this program and we take the opportunity to explain our strategies.

Typical usage of the library TestU01

Typical usage of the library TestU01 involves writing the RNG as a C function and then passing a pointer to this function the TestU01's library interfaces for testing the generator. The purpose of *crush* is to allow any generator written in any language to take advantage of the facitilities of the library TestU01, without requiring the user interested in testing the generator to write the generator in the C programming language.

The chunks of *crush*

The program *crush* is composed of the following chunks. We present them first so you can immediately see its familiar strucutre of a typical C program. However, we do not describe the chunks in the order below because most of these chunks are irrelevant to understanding how the program works.

2 ⟨crush.c 2⟩≡
  ⟨header and declarations 5c⟩
  ⟨a generic generator 3a⟩
  ⟨the main function 3b⟩
  ⟨other functions 5b⟩

Root chunk (not used in this document).

A generic generator

This is the most important part of *crush*. Given the typical usage of TestU01, we would like to implement a generator in C which gets its source of randomness from the **stdin** and not from an arithmetical procedure. By getting the random numbers from the **stdin**, we can feed the program different types of RNG output without having to write another C program for testing against TestU01.

This generator consumes its bytes in binary, so if you're writing a RNG, it should write its random numbers to the **stdout** with, for example, the *fwrite* function from the standard C library. We also assume all data is always written in little-endian.

The bytes will be read into a *buffer* capable of holding 8192 bytes. Since the data we store in this buffer is of type **unsigned int**, this means that the number of **unsigned int** that can fit into this buffer is 8192÷(**sizeof** (**unsigned int**)). We choose 8192 because it is a usual block size for block devices, so, when issuing *fread* calls, we hope to carry much as information we can without using more memory than we must.

Performance is the reason we don't read a single **unsigned int** and return it immediately from the function. That would be too slow. So, the reading logic is as follows. For *buffer* management, we use two variables, *pos* and *limit*. While *pos* is used to mark the next number that the function must return, *limit* divides the buffer into two parts, left and right. To the left of *limit*, there are random numbers read from the **stdin**; to the right, there is empty space, space we didn't use yet: we don't know if the *fread* call will fill up *buffer* completely, so, in such cases, we must know we've read our last valid number and refill *buffer* with another *fread* call.

If *pos* would point to a number that's "on the right side" of *buffer*, than we would produce invalid data. Instead, we must refill the *buffer*.

If we ran out of data from the **stdin**, then *fread* returns 0, in which case we have an error situation. Either the generator has stopped producing data, or some other unexpected situation occurred. We must stop. A generator should be infinite, so, having nothing sensible to do, we stop[1].

[1] This error condition will happen if you're testing randomness from an RNG whose data is in a file on disk and there is not enough data in the file for the chosen battery of tests. The sensible solution is to get a larger file. `BigCrush`, the large battery of tests from the TestU01 library, can consume up to 1428420566888 bytes of data, which is approximately 1330 GiB. If you have very little data, the package *PractRand* [1] is more convenient than the library TestU01 because it incrementally tests your data, using more and more data from your file up until your generator fails a test, when it stops, going up to a maximum of 32 TiB of data.

Having filled *buffer* as much as possible, we can return a random number and increment *pos*. That completes our generic generator. Now that you have read the strategy, you can see it implemented below and read again our description above if necessary.

3a ⟨a generic generator 3a⟩≡

```
unsigned int generator(void)
{
  static uint64_t nbytes;
  static unsigned int buffer[8192 ÷ sizeof (unsigned int)];
  static unsigned int pos; /* where is our number? */
  static unsigned int limit; /* where does the data in the buffer end? */

  if (pos ≥ limit) {
    /* refill the buffer and continue by restarting at 0 */
    limit = fread(buffer, sizeof(unsigned int), (sizeof buffer)÷sizeof(unsigned int), stdin);
    if (limit ≡ 0) {
      // We read 0 bytes. This either means we found EOF or we have
      // an error. A decent generator is infinite, so this should never
      // happen.
      if (ferror(stdin) ≠ 0) {
        perror("fread"); exit(-1);
      }
      if (feof(stdin) ≠ 0) {
        printf("generator produced eof after %ld bytes\n", nbytes);
        exit(0);
      }
    }

    nbytes += limit * sizeof (unsigned int);
    pos = 0;
  }

  unsigned int random = buffer[pos]; /* get one */
  pos += 1;
  return random;
}
```

This code is used in chunk 2.
Defines:
`generator`, never used.

The *main* function

This program is essentially the generic function we implemented above. Everything is straightforward now. In *main*, we just instantiate a generator and run it against a particular battery chosen by the user. The generator is referenced by the structure *unif01_Gen*. It is *unif01_CreateExtenGenBits* that allocates the necessary memory for the generator, but we must call *unif01_DeleteExternGenBits* when we're done, so it can free the allocated resources.

3b ⟨the main function 3b⟩≡

```
int main(int argc, char **argv)
{
  char battery[8]; /* values small, medium or big */
  char name[1000]; /* the name of the generator given by user */
  int option; /* the option produced by getopt() */
  program_name = *argv;

  memset(battery, '\0', sizeof battery);
  memset(name, '\0', sizeof name);

  ⟨option parsing et cetera 4⟩
```

⟨put stdin and stdout in binary mode 5a⟩

unif01_Gen∗ g = unif01_CreateExternGenBits(*name*, *generator*);

if (*strncmp*("small", *battery*, **sizeof** *battery*) ≡ 0) {
bbattery_SmallCrush(*g*);
}
else
if (*strncmp*("medium", *battery*, **sizeof** *battery*) ≡ 0) {
bbattery_Crush(*g*);
}
else
if (*strncmp*("big", *battery*, **sizeof** *battery*) ≡ 0) {
bbattery_BigCrush(*g*);
}
else {
fprintf(**stdout**, "never reached\n");
}

unif01_DeleteExternGenBits(*g*);
return 0;
}

This code is used in chunk 2.

Here's the options we support with sensible user choices checked.

4 ⟨option parsing et cetera 4⟩≡

```
static struct option long_options[] = {
  {"battery", required_argument, 0,  'b' },
  {"name", required_argument, 0,  'n' },
  {"help", no_argument, 0, 'h'},
  {0, 0, 0, 0}
};

int digit_optind = 0;
int option_index = 0;

while ((option = getopt_long(argc, argv, "b:n:h", long_options, &option_index)) ≠ -1)
  switch(option) {
    case 'b': /∗ b as in battery (of tests) ∗/
    if (is_valid_battery(optarg) < 0) {
        fprintf(stdout, "Error: valid batteries are small, medium, big.\n");
        usage();
      }

      /∗ I assume optarg will never be non-null-terminated. ∗/
      strncpy(battery, optarg, sizeof battery);
      break;

    case 'n': /∗ n as in name (of the generator) ∗/
      strncpy(name, optarg, sizeof name);
      break;

    case 'h':
    default:
      usage();
  }
argc -= optind;
argv += optind;

/∗ check if user has given all necessary things ∗/
```

```
if (is_valid_battery(battery) < 0) {
  fprintf(stdout, "Error: valid batteries are small, medium, big\n");
  usage();
}

if (strlen(name) ≡ 0) {
  fprintf(stdout, "Error: you must give a name to your generator\n");
  usage();
}
```

This code is used in chunk 3b.

On Win32 systems, we must explicitly put **stdin** and **stdout** in binary mode. Otherwise, there'll be some byte translations occuring on our backs. For example, if you write byte `'\n'` to the **stdout**, the system will make sure to add byte `'\r'` before writing byte `'\n'` because Win32 systems use the sequence `"\r\n"` as line separator. There seems to be no way of doing this portably.

5a ⟨put stdin and stdout in binary mode 5a⟩≡

```
#ifdef WIN32
  _setmode(_fileno( stdin), _O_BINARY);
  _setmode(_fileno(stdout), _O_BINARY);
#endif
```

This code is used in chunk 3b.

Other chunks

Nothing out of the ordinary here.

5b ⟨other functions 5b⟩≡

```
int is_valid_battery(char *optarg) {
  if (strncmp("small", optarg, sizeof "small") ≡ 0) return 0;
  if (strncmp("medium", optarg, sizeof "medium") ≡ 0) return 0;
  if (strncmp("big", optarg, sizeof "big") ≡ 0) return 0;
  return -1;
}

void usage(void) {
  fprintf(stdout, "Usage: %s [options]\n", program_name);
  fprintf(stdout, " Tests your data for randomness against TestU01.\n\n");
  fprintf(stdout, "Examples:\n");
  fprintf(stdout, "cat /dev/urandom | crush -b small -n 'the local generator'\n");
  fprintf(stdout, "xorshift32 | crush -b small -n xorshift32\n\n");
  fprintf(stdout, " The options are:\n"
  "-b, -battery    Your choice of battery (small, medium, big)\n"
  "-n, -name       The name of your generator\n"
  "-h, -help       Display this information\n");
  exit(0);
}
```

This code is used in chunk 2.
Defines:
`is_valid_battery`, never used.
`usage`, never used.

5c ⟨header and declarations 5c⟩≡

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <getopt.h>
```

```
#ifdef WIN32
#include <fcntl.h> /* _O_BINARY */
#include <io.h>  /* _setmode() */
#endif
#include "TestU01.h"

void usage(void);
int is_valid_battery(char *optarg);
int min(int a, int b);
char *program_name; /* a pointer to argv[0] */
```

This code is used in chunk 2.

References

[1] Chris Doty-Humphrey. "PractRand". In: *URL: http://pracrand.sourceforge.net* (2014).

[2] Donald E. Knuth. "Literate programming". In: *The Computer Journal* 27.2 (1984), pp. 97–111.

[3] Donald E. Knuth and Silvio Levy. *The CWEB system of structured documentation: version 3.0.* Addison-Wesley Longman Publishing Co., Inc., 1994.

[4] Pierre L'Ecuyer and Richard Simard. "TestU01: A C library for empirical testing of random number generators". In: *ACM Transactions on Mathematical Software (TOMS)* 33.4 (2007), p. 22.

[5] Pierre L'Ecuyer and Richard Simard. "TestU01: a software library in ANSI C for empirical testing of random number generators: User's guide, detailed version". In: *Département d'Informatique et de Recherche Opérationnelle Université de Montréal* (2005).

[6] Pierre L'Ecuyer and Richard Simard. *TestU01: A software library in ANSI C for empirical testing of random number generators. Users guide, compact version. (Version: August 17, 2009).*

[7] Norman Ramsey. "Literate programming simplified". In: *IEEE software* 11.5 (1994), pp. 97–105.

Chunk list

Index