Véletlen bit-generátor fejlesztése a NIST SP800-90B dokumentumban foglalt kritériumok alapján

Designing and building a NIST SP 800-90 compliant hardware entropy source

Kutatási jelentés

Dr. Levente Buttyán

2018.07.18.

1. Introduction

In this project, we developed a random number generator that uses hardware entropy sources. The generator's architecture is based on a NIST recommendation (NIST SP800-90), which describes requirements and design specification for such generators. The bulk of the work was carried out by a student, Csongor Ferenczi, and I supervised the project and provided technical guidance. We designed the random number generator to be modular, each module being responsible to a specific task, such as raw data collection from hardware sources, health test of sources, mixing sources, conditioning, entropy estimation, and external interface through which the generator can be called. Our implementation uses camera images as random sources, but thanks to the modularity of our architecture, any kind of random source would be easy to use, and the entire prototype is in general easy to configure and customize. We still have some performance issues that we want to solve in future work.

2. The NIST model

The aforementioned NIST document provides a basic model for the entropy source which can be seen below.

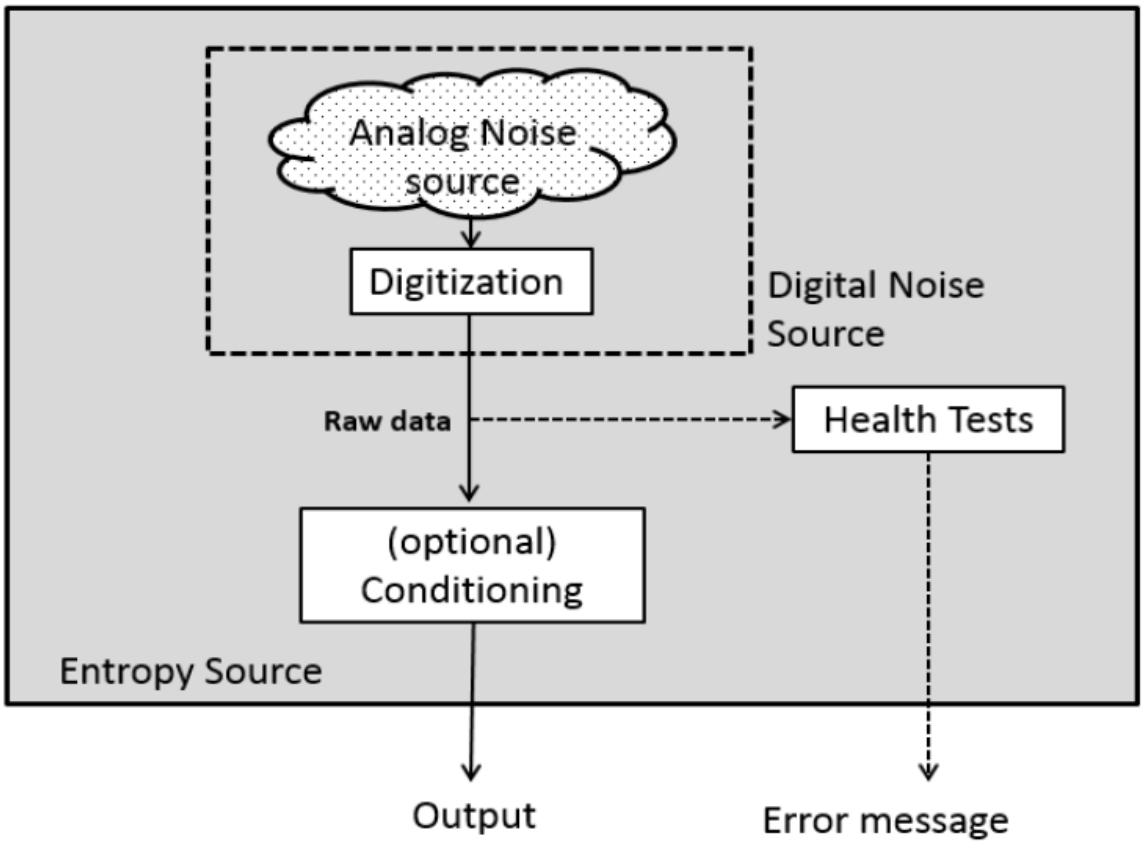

Figure 1: NIST's entropy source model (from Section 2.2)

The core of the entropy source is a noise source which can be digital or analog. A digital noise source can be the quickly changing top part of the RAM, analog noise source can be a thermal noise of a diode for example. Every analog noise source has to be digitalized, to turn the raw samples into bitstrings. Let's call these bitstrings the raw data. We have to continuously run health tests on this raw data in order to make sure that the noise source is in expected working condition. If a health test fails, we have to decide whether the overall entropy source shall be stopped, or it can recover itself. If we cannot serve entropy anymore, we must let the users know this. The raw noise can be optionally conditioned, whose purpose is to remove the bias from the noise. We wanted to make the entropy source more robust, so we decided, that we want to have more noise sources, and we do not want to exclude the use of multiple kinds of noise sources in the future, because we need a high throughput robust entropy source that can scale well and handle even a datacenter's need. For noise sources, we decided to use USB and IP cameras for the prototype.

The NIST document specifies three interfaces (in Section 2.3), which one has to implement: GetNoise, GetEntropy and HealthTest.

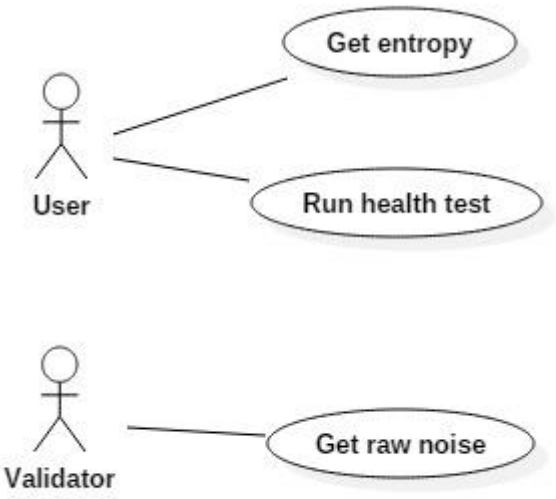

Figure 2: Operating modes and interfaces

The GetNoise interface's task is to provide raw unmodified noise in case we want to validate our solution in a NIST approved laboratory. It is important to note, that this interface is **not** available during regular use. The GetEntropy interface is for providing entropy. The user calls this interface by specifying the amount of entropy they want. The HealthTest interface is for running on demand health tests, so this way the users can make sure that the entropy they got is from a well-functioning noise source.

3. Our implementation of the model

Our prototype was written in Java and its schematic overview can be seen below.

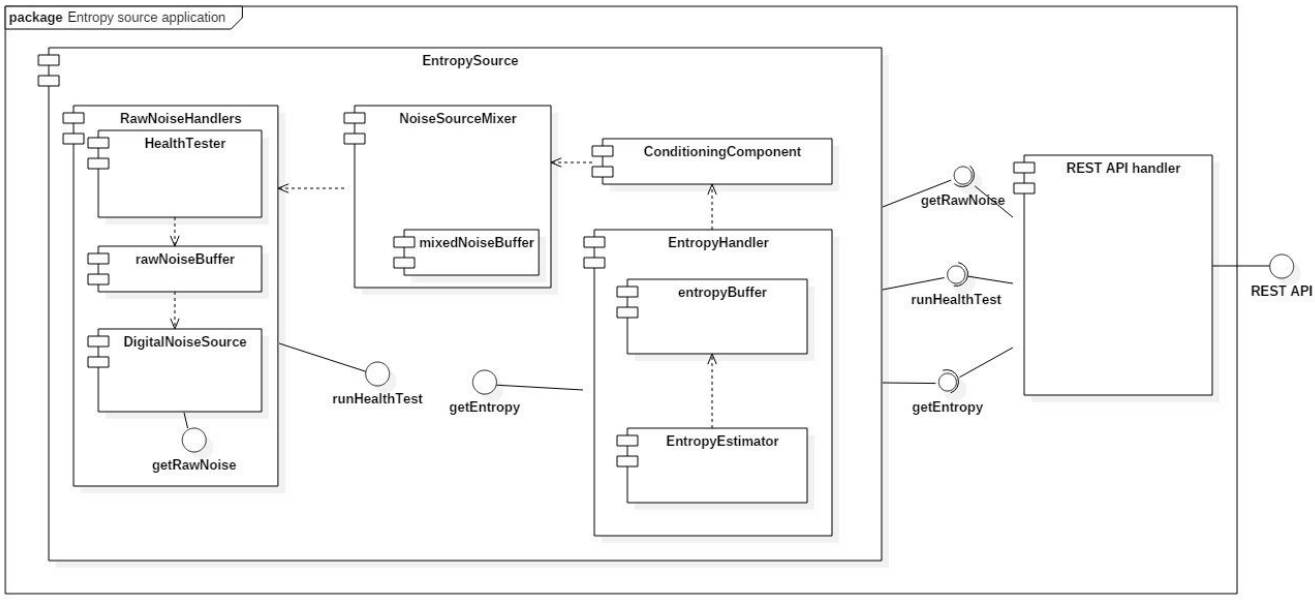

Figure 3: The structure of our implementation

For the sake of illustration, let's assume that currently we use three noise sources: one USB camera and two IP surveillance cameras. Every camera has its own RawNoiseHandler, which can retrieve the frames the camera provides, run health tests on it, and if it passes, buffer the data. Because every camera has a separate handler, if one gets faulty (temporarily or permanently) the other noise sources can still provide entropy for the users. The USB camera we use has a resolution of 640x480 pixel, and an average of 28 fps. Every pixel contains 3 bytes of data, 1 byte for each channel (red, green and blue). This means, that we have about 25.8 MB worth of data every second from this noise source. The IP cameras works with higher resolutions, but lower framerates, so the data throughput is about the same. We implemented three methods to fill the RawNoiseHandler's buffer. The first and fastest method is to simply put the data into the buffer as it arrives. This is very fast, but there can be some really significant repetition in the data stream since pixels next to each other has a higher chance to have similar values. However, as long as there is noise in the frame (which can come from the lighting, sensor noise, traffic, etc.) we still get some randomness. The second method is to fill the buffer only with hashes. Each frame gets hashed, and we only put the hashes into the buffer. This is very slow, and we are throwing away a lot of data (1 MB of raw data becomes 64B of buffered

noise), but this has the highest entropy / byte ratio. The third method is a modified Fisher-Yates shuffling algorithm. We hash the frame, and create a permutation of the frame based on the hash. This method does not give plus entropy to the system, but it does scrambles the data, so the adjacent pixels are not next to each other anymore. The RawNoiseHandlers are connected together by a NoiseSourceMixer, which takes noise from all of the sources, interleaves them together and buffer it (more on this later). After the mixed noise leaves the buffer, it arrives to the ConditioningComponent, which uses a hash algorithm to remove the bias from the noise. This data is then passed to the EntropyHandler which buffers it, and estimates the minimum entropy of the current buffer. The program can communicate with the outside world using a REST API. When a user asks for X amount of entropy, the EntropyHandler can check the active buffer's minimum entropy, and calculate how much data do we have to return in order to provide at least the desired amount of entropy. The data gets outputted in a JSON format. The EntropyHandler uses a multi-buffer system, so this way we can parallelize the minimum entropy estimation.

4. Conclusions

In this project, we designed and implemented a random number generator (entropy source) based on a NIST recommendation (NIST SP800-90). We also integrated into our generator the entropy estimators proposed by NIST and ran performanmce tests both in terms of speed and amount of entropy. We can conclude that the amount of entropy we generate is acceptable even when we use camera images that do not change quickly (but still has some noise). On the other hand, entropy estimation is slow and it is currently a bottleneck in our implementation. At this time, we run only the most conservative estimators to have acceptable speed for the entropy estimation and to be able to serve entropy requests in practical situations. In the future, we plan to optimize some of the entropy estimations to increase speed, and we also want to use other types of random noise sources.