New Entropy Source for Cryptographic Modules Using OpenMP in Multicore CPUs

Jungbai Kim, Taeill Yoo, Yongjin Yeom and Okyeon Yi

Department of Mathematics, Cryptography & Information Security Institute, Kookmin University, Seoul, Korea

{kimjb1985, taeillyoo, salt, oyyi}@kookmin.ac.kr

Abstract

Generating random numbers with high entropy plays an important role in cryptography. The security of a modern cryptosystem can be proved under the assumption that the system uses a reliable random number generator in general. However, it is difficult to implement a module with sufficiently qualified entropy sources in its random number generators. With trends to move to lightweight devices from desktop PCs, cryptographic modules have suffered from the lack of good sources of randomness. As another trend, parallel computing techniques have become popular very rapidly to achieve better performance. In this paper, we suggest a method for generating entropy source using parallel computations with OpenMP. Also, we verify the experimental results based on the min-entropy estimation and conclude that a new source of randomness can be obtained from the race conditions in multicore environments.

Keywords: *Entropy source, Random number generation, Cryptographic modules, Parallel program, Race condition, OpenMP*

1. Introduction

In recent decades, random number or pseudo random number has been used in several parts of IT, particularly in the field of information security and cryptography. To generate encryption key or to establish key agreement in communication protocols, random numbers are essential to make them secure. If the output of a random number generator is predictable in a cryptosystem, the system cannot reach the intended level of security. The input to the deterministic part of random number generators is called the '*seed*' which is generated by entropy sources. Once the seed is exposed or guessed with non-negligible probability to an adversary, the random number generator cannot be secure any more. Therefore, it is important to keep seeds secret as well as to collect sufficient entropy from the source of randomness.

In the past, most cryptographic modules have been developed and used in desktop or server computers in which we can use abundant physical sources of randomness. However, according to the rapid change to smart daily life with smart devices, we have to consider the environments for cryptographic modules in lightweight devices, where it is hard to find sources of randomness with high entropy.

1.1. Entropy source

In most cryptographic modules, in order to operate their random bit generators properly, entropy sources have to provide sufficient randomness. The studies of entropy source have

been progressed for modules working at desktop environments. However, it is very difficult to find good entropy sources in lightweight devices. For example, in most embedded devices, there are no keyboard, mouse and hard disk which provide random noise as entropy sources to cryptosystems. Accordingly, finding better entropy source has become more important. To evaluate and verify entropy sources, we can refer a document SP 800-90B by NIST. It consists of the recommendation for entropy sources used for random bit generation.

1.2. Parallel computing

The size of data that a computer has to process has increased very rapidly. However, the degree of integration of single core CPU and increasing CPU clock reaches the limit. In order to overcome this problem, the shift toward multi core architectures has been considered as an alternative.

A multi core CPU is an integration of more than one independent cores in a single package. It is common to use multi core CPU such as dual-core, quad-core, and octa-core, etc. To use their computing powers efficiently, we can adopt OpenMP for parallel programming. If we use a PC with GPU which supports general purpose computations, called GPGPU, we can make use of GPU as a parallel computing resource through tools such as CUDA and OpenCL. Comparing parallel computing in GPU with that in multi core CPU, the latter is easier to handle with OpenMP and does not require additional hardware in a PC. To sum up, parallel computing has become widely accepted and supporting tools have been developed actively.

1.3. Organization of the paper

The rest of the paper is organized as follows: In Section 2 several results developed by previous researches on generating random noise will be summarized. Section 3 proposes our idea to generate random noise for providing entropy source to cryptographic modules. In Section 4 we analyze our implementations on several multi core environments and finally, we wrap up and conclude our results in Section 5.

2. Related Work

2.1. Generating Entropy Source in a PC

Lack of entropy sources. Dorrendorf *et al.*, [4] found several weaknesses of the random number generator in Windows 2000 by reverse engineering the binary code without any help from Microsoft. One of main vulnerabilities is that the difficulty of extracting sufficient entropy. After their paper was published, Microsoft recognized that Windows XP also has the same vulnerability as Windows 2000 in the random number generator. Linux also suffers from the lack of entropy source as pointed out by Gutterman *et al.*, [13].

Use of physical entropy source. Usually, random bits used in cryptographic modules can be obtained from the noise sources in their environment. As external sources in a usual PC, we may use mouse and keyboard activity, disk In/Out, and specific interrupt, *etc*. When a certain event happens on one of them, it generates data of 32-bit size. As for the mouse, the word is generated to indicate timing when the mouse is clicked, and data from the keyboard indicates that encoded the attributes of the keyboard (*e.g.*, a button which was pressed). Though there are several physical entropy sources available in a PC, it is not sufficient to provide random bits with enough entropy for cryptographic modules. In embedded systems, the situation is even worse because they have less physical devices.

2.2. Generating Random Number and Histogram Equalization Techniques

Recently, several methods for generating random noise using GPU were proposed by Chan *et al.*, [1] and Yeom [11]. They used the uncertainty caused by massively parallel computations. After extracting random noise, they applied post-processing called histogram equalization technique to obtain bit sequences of high entropy. Histogram equalization is a process on an image to improve its contrast. When an image is too dark or too bright, this technique is useful to adjust it so that the intensities can be better distributed on the histogram. Biased entropy noise can be adjusted by histogram equalization to be closer to the uniform distribution.

3. Generating Random Noise

3.1. Race Condition

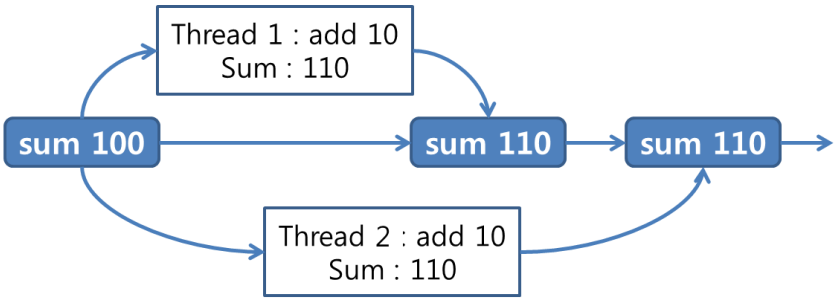


Figure 1. Race Condition on shared memory caused by simultaneous updates [12]

In parallel computing, an event in which more than one threads try to use the same resource simultaneously is called a *race condition*. When many processors or threads access the same resource competitively, the order of executions is not determined clearly. Consequently, there is a risk that the result will be different from the programmer's intention. Figure 1 shows an example of a race condition caused by adding 10 to the same memory address in parallel.

3.2. Observations on Race Conditions

In order to generate random noise, we make use of race conditions in multi core CPU using OpenMP. That is, we raise a race condition on purpose and save the result which seems to be unpredictable and non-deterministic. Algorithm 1 describes parallel data updates by incremental operation (++) in shared memory.

Algorithm 1. Basic code which generates race conditions

```
#pragma omp parallel num_threads(2)
    {
#pragma omp for
          for(int i=0; i< N; i++)
                Sum++;
    }
```

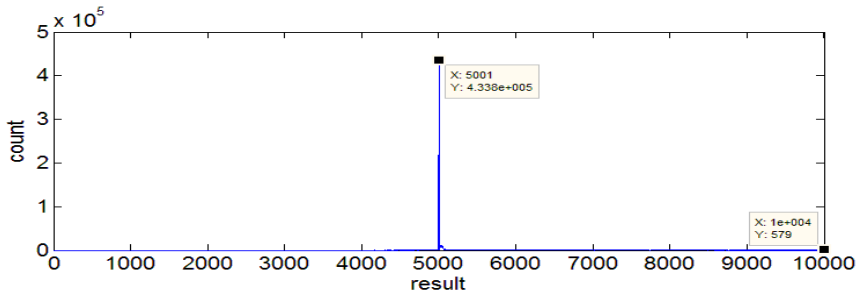


Figure 2. The distribution of the variable Sum produced by one million executions of Algorithm 1 with N=10,000

If all updates are successfully finished without race conditions, the variable `Sum` holds `N` at the end of Algorithm 1. However, if race conditions happen at the significant amount of updates, we expect that the result is smaller than N. Figure 2 shows the distribution of the results stored in the variable Sum when we execute one million times of Algorithm 1 with the number of iterations N=10,000. Approximately, most of the results are found around N/2, which means half of updates were successful. Even though Algorithm 1 brings about race conditions, the results are not distributed widely. Therefore, it is pretty possible to predict the results which cannot be suitable yet for a reliable source of randomness.

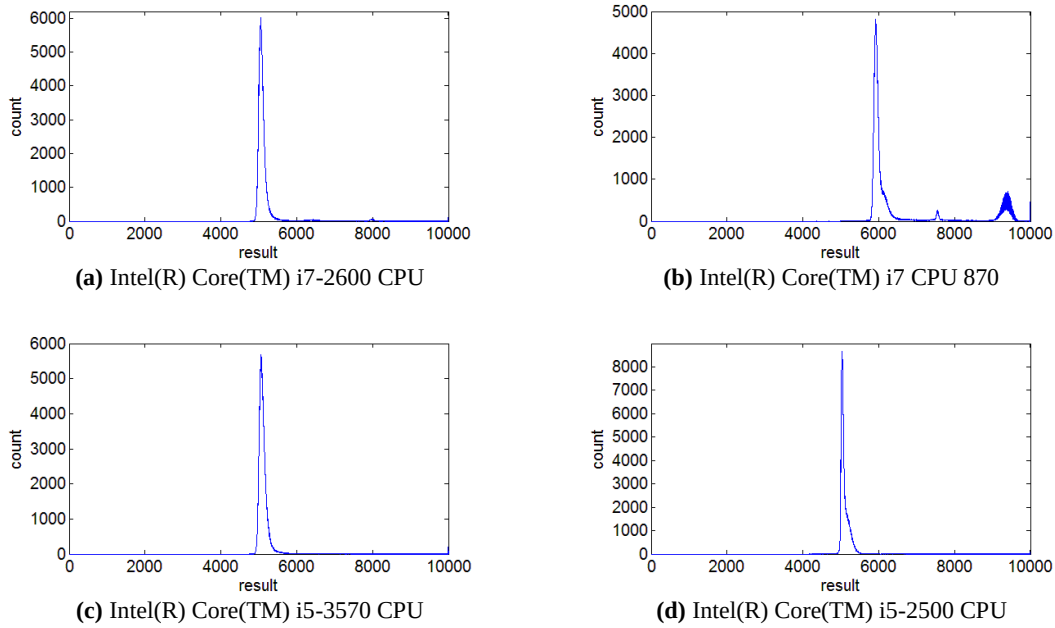
3.3. Advanced Experimental Race Condition

After trying to improve Algorithm 1 in several ways in order to obtain better distributions suitable for entropy sources, we finally construct Algorithm 2 which uses modulo operation on two arrays. Suppose that arrays `InputA` and `InputB` are initialized with sequences (1^2, 2^2, 3^2, …) and (1, 2, 3, …), respectively. Then modulo operations with the i-th element `InputA`[i] of `InputA` and the (i-1)-th element `InputB`[i-1] of `InputB` produce 1 for i=2,3,4, …, since i^2 % (i-1) = 1.

Algorithm 2. Generating random noise based on race conditions using modulo operations

```
#pragma omp parallel num_threads(2)
    {
#pragma omp for
          for(int i=1; i<= N; i++)
                Sum +=InputA[i] % InputB[i-1]
    }
```

We perform Algorithm 2 on several multi core CPUs. Figure 3 shows the resulting distribution of the variable `Sum` after executing Algorithm 2 one million times with `N`=10,000. Comparing with Figure 2, we obtain the distributions of wider range similar to a Chi-square distribution on several multi core environments.



(a) Intel(R) Core(TM) i7-2600 CPU

(b) Intel(R) Core(TM) i7 CPU 870

(c) Intel(R) Core(TM) i5-3570 CPU

(d) Intel(R) Core(TM) i5-2500 CPU

Figure 3. The distribution of the variable Sum produced by one million executions of Algorithm 2 with N=10,000 on several multi core environments

4. Experimental Results

The resulting distributions depend upon the environments. Some results have more than one peak (See (b) of Figure 3). In order to extract data with high entropy, we truncate data far from the largest peak. Selected data as in Figure 4, are not suitable yet used as seeds for random number generators because they are still far from the uniform distribution. Therefore we have to consider the following two additional steps.

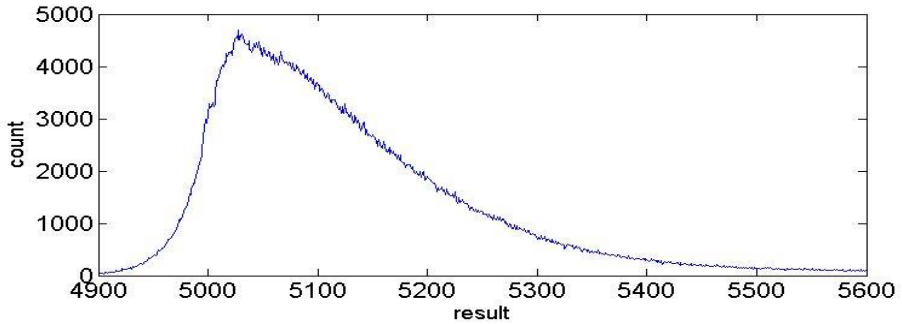


Figure 4. The distribution of selected data around the largest peak (*x*-axis: resulting values, *y*-axis: frequency counts of the value)

In the first step, we apply the histogram equalization method. Selected data can be depicted as Figure 4 which is, in fact, an enlarged view of Figure 3. If we observe each result in every single execution of Algorithm 2, the output of the algorithm shows random noise as in Figure 5. We divide the range of the possible output into several bins so that the probabilities for partitions are almost equal to each other. It is convenient to choose the number of bins as the

powers of 2. If we use 2, 16, and 256 bins, we may assign bit, nibble, and byte for each bin, respectively. For each execution, the result can be interpreted as a bin index and converted into its corresponding value. For example, if we use 2 bins, then each execution of Algorithm 2 generates a single bit.

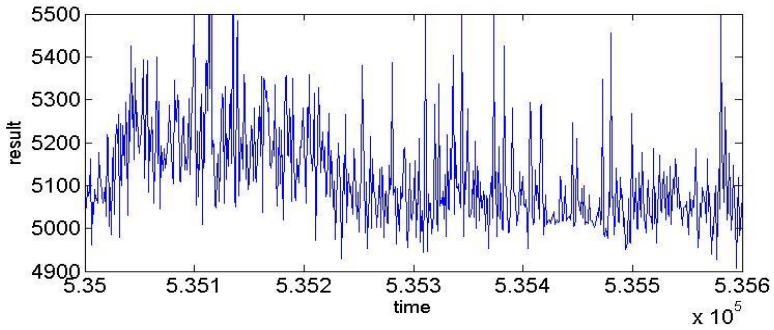


Figure 5. The result stored in the variable Sum at each execution of Algorithm 2 (x-axis: trial counter, y-axis: results in the variable Sum)

There should be one more step in which we verify whether the output of the first step is suitable as a source of randomness. To find the lower bound of entropy, we estimate min-entropy on the output of the histogram equalization in the first step. We perform the procedure defined as 'Estimating the Min-Entropy of IID Sources' in SP800-90B [3].

Table 1 shows that the output has almost full entropy. In the last column for Intel(R) Core(TM) i5-2500, we collect 6 million bytes (5.72 MB) and obtain about 500 KB after histogram equalization using 2 bins. The lower bound of entropy estimation for 1 bit is 0.9996. If we use 16, 256 bins, then estimated min-entropy would be 3.9900, 7.9331, out of 4 and 8, respectively. If we choose larger number of bins, we can extract more entropy from the same source.

Table 1. Result of Estimating the Min-Entropy in four different CPUs

The number of bins (bit)	Intel(R) Core(TM) i7-2600 CPU	Intel(R) Core(TM) i7 CPU 870	Intel(R) Core(TM) i5-3570 CPU	Intel(R) Core(TM) i5-2500
2(1 bit)	0.9982	0.9997	0.9993	0.9996
16(4 bit)	3.9904	3.9900	3.9899	3.9900
256(8 bit)	7.9361	7.9366	7.9347	7.9331

5. Conclusion

In this paper, we propose a method to generate random noise which can be used as input to a random number generator. To overcome the lack of entropy sources for cryptographic modules, we use race conditions caused by parallel computations in multi core CPUs. By repeating competitive updates on the shared memory, we could obtain noisy data as entropy source. After that, histogram equalization technique is applied to extract a sequence of bits with high entropy. We verify the results on several multi core CPUs. Also, we measure min-

entropy for each data which shows that the output has sufficient entropy as a source of randomness.

Acknowledgements

This work was supported by the IT R&D program of MOTIE/KEIT. [10039140, Development of Crypto Algorithms (ARIA, SEED, KCDSA, *etc.*) for Smart Devices (ARM7/9/11, UICC)]

References

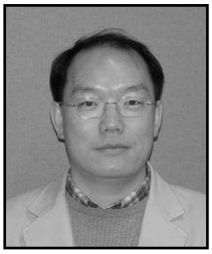
[1] J. Chan, B. Sharma, c. G. Jiaqing Lv, R. Thulasiram and P. Thulasiraman, "True Random Number Generator Using GPUs and Histogram Equalization Techniques", High Performance Computing and Communications (HPCC), IEEE 13th International Conference, **(2011)**.
[2] C. Lauradoux, J. Ponge and A. Roeck, "Online Entropy Estimation for Non-Binary Sources and Applications on iPhone", INRIA/RR—7663, **(2011)**.
[3] E. Barker and J. Kelsey, "NIST DRAFT Special Publication 800-90B, Recommendation for the Entropy Sources Used for Random Bit Generation", National Institute of Standards and Technology, **(2012)**.
[4] L. Dorrendorf, Z. Gutterman and B. Pinkas, "Cryptanalysis of the Random Number Generator of the Windows Operating System", ACM Transactions on Information and System Security, vol. 13, no. 1, Article 10, **(2009)**.
[5] M. Isard and Y. Yu, "Distributed data-parallel computing using a high-level programming language", International Conference on Management of Data (SIGMOD), **(2009)**.
[6] M. J. Jun and J. J. Lee, "Image Histogram Equalization Based on Gaussian Mixture Model", Journal of Korea Multimedia Society, vol. 15, no. 6, **(2012)**.
[7] OpenMP Application Program Interface: Public Review Release Candidate 1, Version 4.0 RC 1, **(2012)** November.
[8] P. Hagerty and T. Draper, "Entropy Bounds and Statistical Tests", Random Bit Generator Workshop, **(2012)** December.
[9] S. Carr, J. Mayo and C. -K. Shene, "Race conditions: a case study", Journal of Computing Sciences in Colleges, vol. 17, Issue 1, **(2001)**.
[10] Y. Yu, P. K. Gunda and M. Isard, "Distributed Aggregation for Data-Parallel Computing Interfaces and Implementations", Proceeding SOSP '09 Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, **(2009)**, pp. 247-260.
[11] Y. Yeom, "Generating Random Numbers for Cryptographic Modules Using Race Conditions in GPU", T. -h. Kim *et al.*, (Eds.), GDC/IESH/CGAG, CCIS vol. 351, Springer, **(2012)**, pp. 96-102.
[12] Y. Jeong, "OpenMP Parallel Programing", Freelac (in Korean) **(2011)**.
[13] Z. Gutterman, B. Pinkas and T. Reinman, "Analysis of the Linux Random Number Generator", Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06), **(2006)**.

Authors

Jungbai Kim received his B.S. degree in Mathematics from Kookmin University, Korea, in 2012. He is currently undertaking an M.S. course in Mathematics from Kookmin University, Korea. His current research interests are in the areas of parallel computing, cryptography and information security.

Taeill Yoo received his B.S. degree in Mathematics from Kookmin University, Korea, in 2010. He is currently undertaking an M.S. course in Mathematics from Kookmin University, Korea. His current research interests are in the areas of parallel computing, cryptography and information security.

Yongjin Yeom received his B.S., M.S., and Ph.D. degrees in Mathematics from Seoul National University, Korea, in 1991, 1994, and 1999, respectively. He worked at the attached institute of ETRI (*aka* NSRI) from 2000 to 2011. Since 2012, he has been working at the department of Mathematics in Kookmin University, where he is currently an assistant professor. His research interests are design and analysis of cryptographic algorithms, and parallel implementations.



Okyeon Yi received his B.S. and M.S. degrees from Korea University in 1988 and 1990, respectively, and received Ph.D. degree in Mathematics from University of Kentucky, USA, in 1996. After working at the Electronics & Telecommunications Research Institute (ETRI) from 1999 to 2001, he joined Kookmin University in 2001, where he is currently an Associate Professor of the Mathematics. His interests are Information Security, Mobile Communication and Cryptography.