Generating Entropy Source for Cryptographic Modules Using OpenMP in Multicore CPUs
(Extended Abstract)

Jungbai Kim, Taeill Yoo, Yongjin Yeom, Okyeon Yi
Department of Mathematics, Kookmin University, Seoul, Korea
{kimjb1985, taeillyoo, salt, oyyi}@kookmin.ac.kr

Abstract. In cryptography, entropy plays a significant role, particularly, when the developers implement cryptographic modules. However, with trends to change to light device from desktop, it is difficult to collect sufficient entropy for Cryptographic Modules. As another trend, many parallel techniques have been considered to deal with massive data in many fields of Information Technology. In this paper, we propose a method of generating entropy using one of parallel computing, OpenMP, in cryptographic modules.

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 and 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 provide sufficient level of security. The input to deterministic random number generator is called the ‘seed’ which is generated by entropy sources. Once the seed can be 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 provide sufficient entropy from noise source.

In the past, most cryptographic modules have been developed and used in desktop or server computers. 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.

2 Generating Random Noise

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.

In order to generate random noise for seed, 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 updates data by incremental operation (+++) in shared memory. If all updates are successfully finished without race conditions, the variable Sum holds N at the end of Algorithm 1.

Algorithm 1. Basic code which generates race conditions

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

If a race condition happens at significant amount of updates, we expect that the result is smaller than N. Conversely, if there are no race conditions during the simultaneous updates, we expect the variable Sum holds N. 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 random noise for entropy source.

In order to solve the problem, we improve Algorithm 1 in several ways to find a suitable distribution for entropy. One of the several ways results in better distribution as the left of Fig 1.

Fig. 1.Left : The distribution of selected data around the largest, Right : The result stored in the variable Sum at each execution

3 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, we use race conditions caused by parallel computations in multi core CPUs.
As a future work, the basic strategy should be improved more efficiently. Additionally, we have to consider post processing to obtain random bits with high entropy. Also, to verify the final output, test suit for entropy sources in SP800-90B should be applied.

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