AMD

AMD Secure Random Number Generator Library

Publication # **56310** Revision: **0.70**
Issue Date: **April 2018**

Advanced Micro Devices

Contents

List of Figures

List of Tables

Revision History

Date	Revision	Description
April 2018	0.70	Initial public release.

Introduction

Random numbers and their generation is a crucial component in many areas of computational science. Examples include Monte Carlo simulations, modeling, cryptography, games and many more. One of the most significant fields where random numbers are used is cryptography. Cryptographic applications require random numbers for key generation and encrypting and decrypting content. To be considered secure, these cryptographic applications need random numbers that are unpredictable and robust.

There are mainly two types of random number generators (RNG); software-based pseudorandom number generators (PRNG) and hardware random number generators. Pseudorandom number generators use a mathematical model and are implemented in software. Hardware random number generators, on the other hand, rely on a dedicated hardware unit to generate random numbers.

- **Pseudorandom number generators –** Pseudorandom number generators follow a deterministic approach and an algorithmic implementation of a mathematical model. Depending on the initial state or seed, they generate a sequence of random numbers. While the generated numbers may have good statistical properties, being statistically independent and having a uniform distribution, they are vulnerable to attacks once the seed is determined. If someone can guess the initial state of the generator and the mathematical model, they will be able to calculate the rest of the sequence of random numbers. Thus, pseudorandom number generators are not a wise choice for cryptographic applications.

- **Hardware random number generators –** Hardware random number generators differ by providing more unpredictable random numbers. They generate random numbers utilizing a hardware unit as a source of randomness, unlike a fixed mathematical model used in a PRNG. Determining the pattern of random numbers from a physical source is more difficult and are more suitable for cryptographic applications. The AMD "Zen" microprocessor architecture is equipped with a cryptographic coprocessor that enables generation of cryptographically secure random numbers.

This document provides details on AMD Secure Random Number Generator (Secure RNG) library which provides easy-to-use software Application Programming Interface (API) to access random numbers generated by AMD hardware RNG implementation.

AMD Hardware RNG Architecture

Processors based on AMD Zen are equipped with a cryptographic coprocessor that enables the generation of secure random numbers. AMD hardware RNG architecture is shown in Figure 1 on page 7.

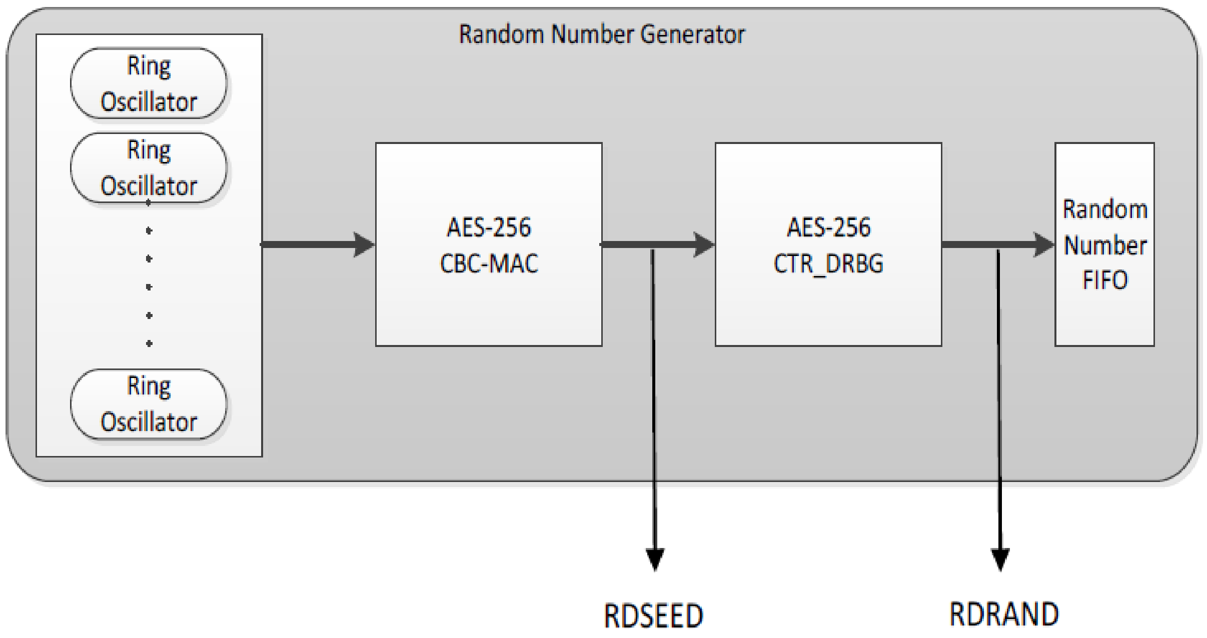

Figure 1. AMD Hardware RNG Architecture

The main components are

- **Noise Source –** The RNG uses 16 separate ring oscillators chains as the noise source. During runtime, the 16 ring oscillators are continually sampled to generate noise values.
- **Entropy Conditioner –** The 16 bits of ring oscillator noise are fed into the entropy conditioner, based on AES-256[i] CBC-MAC[ii], which gathers multiple noise samples over time to use in generating the randomness needed by the RNG design. To create a seed value, three iterations are executed, each producing a 128-bit random number, thus generating a total 384-bit seed. The seed is then fed into the deterministic random bit generator, AES-256 CTR_DRBG module.
- **Deterministic Random Bit Generator (DRBG) –** AMD RNG design includes a deterministic random bit generator, based on AES-256 CTR_DRBG. It uses the 384-bit seed value from the entropy conditioner and produces random values quickly. The values are stored in a FIFO buffer to support fast read bursts. A maximum of 2048 32-bit samples are generated per seed by the DRBG module. It attempts to aggressively re-seed well before this limit is reached. The DRBG module is compliant with NIST SP 800-90A standard for random bit generation.

AMD Secure RNG Library

The AMD hardware RNG design allows access to output registers that enable reading the random values generated by the hardware. These registers are accessible to software through an x86 user-level instruction. The x86 instructions are:

- RDRAND – Returns a 16-bit, 32-bit, or 64-bit random value
- RDSEED – Returns a 16-bit, 32-bit, or 64-bit conditioned random value

Accessing the random values using these low-level instructions can be cumbersome in high-level applications. Also, most applications would need a stream of random numbers which means multiple calls to RDRAND/RDSEED instructions.

AMD Secure RNG library provides an easy-to-use API library for accessing the hardware generated random numbers. The use of the library has several advantages:

- Applications can link to the library and invoke either a single or stream of random numbers
- It abstracts out low-level programming for accessing RDRAND and RDSEED instructions as well as handling some of the possible outcomes based on register outputs
- The library also manages checking for a hardware failure while generation and allows the user to specify retrial attempts
- The intuitive API interfaces enables more applications to use the library and thus the underlying AMD RNG implementation

The Secure RNG library exposes several APIs that makes use of the above instructions to either return a single random value or a stream of them. The APIs and their functionality are described in Table 1.

Table 1. AMD Secure RNG APIs

Hardware Support APIs	
int is_RDRAND_Supported()	
Description:	Checks for RDRAND instruction support
Parameters:	None
Return type:	*int*: indicates whether RDSEED is supported or not. 1 - Success
int is_RDSEED_Supported()	
Description:	Checks for RDSEED instruction support
Parameters:	None
Return type:	*int*: indicates whether RDSEED is supported or not. 1 - Success
RDRAND APIs	
*int get_rdrand16u(uint16_t *rng_val, unsigned int retry_count)*	
Description:	Fetch a single 16-bit value by calling the RDRAND instruction
Parameters:	rng_val – Pointer to memory to store the value returned by RDRAND retry_count – Number of retry attempts
Return type:	*int*: Success or failure status of function call
*int get_rdrand32u(uint32_t *rng_val, unsigned int retry_count)*	
Description:	Fetch a single 32-bit value by calling the RDRAND instruction
Parameters:	rng_val – Pointer to memory to store the value returned by RDRAND

	retry_count – Number of retry attempts
Return type:	*int*: Success or failure status of function call
*int get_rdrand64u(uint64_t *rng_val, unsigned int retry_count)*	
Description:	Fetch a single 64-bit value by calling the RDRAND instruction
Parameters:	rng_val – Pointer to memory to store the value returned by RDRAND retry_count – Number of retry attempts
Return type:	*int*: Success or failure status of function call
*Int get_rdrand32u_arr(uint32_t *rng_arr, unsigned int N, unsigned int retry_count)*	
Description:	Fetch an array of 32-bit values of size N by calling the RDRAND instruction
Parameters:	rng_arr – Pointer to memory to store the value returned by RDRAND N – Number of random values to return retry_count – Number of retry attempts
Return type:	*int*: Success or failure status of function call
*int get_rdrand64u_arr(uint64_t *rng_arr, unsigned int N, unsigned int retry_count)*	
Description:	Fetch an array of 64-bit values of size N by calling the RDRAND instruction
Parameters:	rng_arr – Pointer to memory to store the value returned by RDRAND N – Number of random values to return retry_count – Number of retry attempts
Return type:	*int*: Success or failure status of function call
*Int get_rdrand_bytes_arr(unsigned char *rng_arr, unsigned int N)*	
Description:	Fetch an array of random bytes of a given size by calling the RDRAND instruction
Parameters:	rng_arr – Pointer to memory to store the random bytes N – Number of random bytes to return
Return type:	*int*: Success or failure status of function call
RDSEED APIs	
*int get_rdseed16u(uint16_t *rng_val, unsigned int retry_count)*	
Description:	Fetch a single 16-bit value by calling the RDSEED instruction
Parameters:	rng_val – Pointer to memory to store the value returned by RDSEED retry_count – Number of retry attempts
Return type:	*int*: Success or failure status of function call
*int get_rdseed32u(uint32_t *rng_val, unsigned int retry_count)*	
Description:	Fetch a single 32-bit value by calling the RDSEED instruction
Parameters:	rng_val – Pointer to memory to store the value returned by RDSEED retry_count – Number of retry attempts
Return type:	*int*: Success or failure status of function call

*int get_rdseed64u(uint64_t *rng_val, unsigned int retry_count)*	
Description:	Fetch a single 64-bit value by calling the RDSEED instruction
Parameters:	rng_val – Pointer to memory to store the value returned by RDSEED retry_count – Number of retry attempts
Return type:	*int*: Success or failure status of function call
*int get_rdseed32u_arr(uint32_t *rng_arr, unsigned int N, unsigned int retry_count)*	
Description:	Fetch an array of 32-bit values of size N by calling the RDSEED instruction
Parameters:	rng_arr – Pointer to memory to store the value returned by RDSEED N – Number of random values to return retry_count – Number of retry attempts
Return type:	*int*: Success or failure status of function call
*int get_rdseed64u_arr(uint64_t *rng_arr, unsigned int N, unsigned int retry_count)*	
Description:	Fetch an array of 64-bit values of size N by calling the RDSEED instruction
Parameters:	rng_arr – Pointer to memory to store the value returned by RDSEED N – Number of random values to return retry_count – Number of retry attempts
Return type:	*int*: Success or failure status of function call
*int get_rdseed_bytes_arr(unsigned char *rng_arr, unsigned int N)*	
Description:	Fetch an array of random bytes of a given size by calling the RDSEED instruction
Parameters:	rng_arr – Pointer to memory to store the random bytes N – Number of random bytes to return retry_count – Number of random bytes to return
Return type:	*int*: Success or failure status of function call

The APIs, *is_RDRAND_supported* and *is_RDSEED_supported* use CPUID instruction to check support for RDRAND and RDSEED instructions respectively. Applications should initially invoke these APIs to verify hardware support of Secure RNG. The code snippet below shows sample usage of the library API to return an array of 1000 64-bit random values using RDRAND

```
//Check for RDRAND instruction support
int ret = is_RDRAND_supported();
int N = 1000;

//If RDRAND supported
if(ret == SECRNG_SUPPORTED)
{
   uint64 t rng64;

   //Get 64-bit random number
   ret = get_rdrand64u(&rng64,0);

   if(ret == SECRNG_SUCCESS)
           printf("RDRAND rng 64-bit value %lu\n\n", rng64);
```

```
      else
               printf("Failure in retrieving random value using RDRAND!\n");
      //Get a range of 64-bit random values uint64_t*
      rng64 arr = (uint64 t*) malloc(sizeof(uint64 t) * N);
      ret = get_rdrand64u_arr(rng64 arr, N, 0);
      if(ret == SECRNG SUCCESS)
               printf("RDRAND for %u 64-bit random values succeeded!\n", N);
      else
               printf("Failure in retrieving array of random values using
               RDRAND!\n");
    }
    else
    {
               printf("No support for RDRAND!\n");
    }
```

Applications

Applications relying on random numbers are innumerable. Many high-performance computing (HPC) applications including Monte Carlo simulations, communication protocols and gaming applications depend on random numbers. One of the ubiquitous use of unpredictable random numbers is in cryptography. It underlies the security mechanism of modern communication systems such as authentication, e-commerce, etc.

The key applications of random number generators in the field of cryptography and internet security are

- Key generation operations
- Authentication protocols
- Internet gambling
- Encryption
- Seeding software based pseudo-random number generators (PRNG)

Zen has a strong focus on the cryptography domain with a dedicated hardware block, the cryptographic co-processor, found in the AMD Secure Processor. The AMD Secure RNG library provides a suite of APIs which developers can easily use in any of their applications.

Secure RNG Performance

Every time you invoke the RDRAND or RDSEED instruction it reads a value from the cryptographic co-processor block. The read operation, being a memory-mapped I/O (MMIO)[iii] access, is relatively slow compared to other software-based PRNGs. However, in cryptography applications that are not latency sensitive, this may not be much of an issue. For latency sensitive cases, a hybrid approach could be used where a PRNG implementation can be seeded using a value from the hardware generated random value/seed. This approach would provide a higher degree of security as well as improve performance.

Conclusion

Hardware-based random number generators provide more secure and robust random values than software implemented generators. Processors based on AMD Zen include a random number generator design in its hardware. The software access to these modules is quite low-level with MMIO and x86 instructions. The AMD Secure RNG library abstracts most of the low-level instruction calls and provides an easy to use API interface. Applications such as cryptographic key generation and many others can benefit from the library and have access to more random numbers.

---

[i] Recommendation for Random Number Generation Using Deterministic Random Bit Generators *http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf*.

[ii] CBC-MAC *https://en.wikipedia.org/wiki/CBC-MAC*.

[iii] Memory-mapped I/O *https://en.wikipedia.org/wiki/Memory-mapped_I/O*.