AMD Secure Random Number Generator Library

INTRODUCTION

Random numbers and their generation is a crucial component in many areas of computational science. Monte Carlo simulation, modeling, cryptography, games and many more. One of the vital fields where random numbers are used is Cryptography. Cryptographic applications require random numbers for key generation, encrypting messages or to mask certain content. In order to be secure, they need high quality random numbers that must be 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.

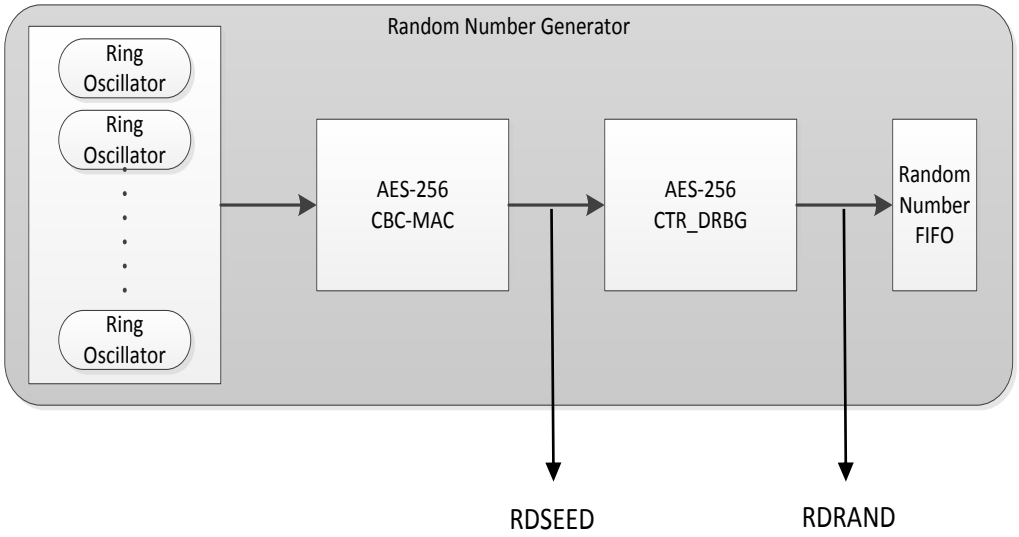
- **Pseudorandom number generators:** Pseudorandom number generators follow a deterministic approach and an algorithmic implementation of a mathematical model. Depending on the initial state (seed), they generate a sequence of random numbers. While they may have good statistical properties (statistically independent and uniform distribution), they are vulnerable to state compromise attacks once an adversary becomes aware of the seed. If one is able to guess the initial state of the generator and the mathematical model, they will be able to calculate rest of the sequence of random numbers. Hence pseudorandom numbers are not a good choice for cryptographic applications.
- **Hardware random number generators:** Hardware random number generators provide more unpredictable random numbers. They generate random numbers utilizing a hardware unit as a source of entropy (randomness) unlike a fixed mathematical model in PRNG. Determining pattern of random numbers from such a physical source is much harder and hence have more suitable cryptographic properties. AMD's latest microprocessor architecture are equipped with Cryptographic Co-processor hardware that enables generation of cryptographically secure random numbers.

In this whitepaper, we provide details on AMD Secure Random Number Generator (Secure RNG) library which provides an easy-to-use software Application Programming Interface (API) to access random numbers generated by AMD's hardware RNG implementation.

AMD HARDWARE RNG ARCHITECTURE

AMD's Family 17h processor and other products of 2016 and beyond are equipped with Cryptographic Co-processor 5.0 hardware that enables the generation of cryptographically secure random numbers.

AMD's hardware RNG architecture is shown in the figure below



source:AMD RNG Whitepaper by David Kaplan[3]

Figure 1. AMD Hardware RNG Architecture

The main components are

1. **Noise Source**: The RNG uses 16 separate ring oscillator chains as the noise source. During runtime, the 16 ring oscillators are continually sampled to generate noise values.
2. **Entropy Conditioner**: The 16-bits of ring oscillator noise are fed into the entropy conditioner, based on AES-256[1] CBC-MAC[2], which gathers multiple noise samples over time to use in generating the entropy needed by the RNG design. To create a seed value, 3 iterations are executed, each generating 128 bits of full entropy, thus generating a 384-bit seed. The seed is then fed into the deterministic random bit generator, AES-256 CTR_DRBG module.
3. **Deterministic Random Bit Generator (DRBG)**: AMD RNG design includes a deterministic random bit generator, based on AES-256 CTR_DRBG construct. It uses the 384-bit seed value from the entropy conditioner and produces fast, high-quality random values. The values are stored in a FIFO buffer to support fast read bursts. 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 [1] for random bit generation.

AMD Secure RNG Library

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

1. **RDRAND** : Returns a 16-bit, 32-bit or 64-bit random value
2. **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 library has several advantages

- Applications can just 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.
- Library also manages checking for any hardware failure while generation and allows 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 is as described in the table below.

Feature	API	Description
Hardware Support	int is_RDRAND_Supported( )	Check RDRAND instruction support Parameters: None Return type *int*: Indicates whether RDRAND is supported or not. 1 – Success
	int is_RDSEED_Supported( )	Check RDSEED instruction support Parameters: None Return type *int*: Indicates whether RDSEED is supported or not. 1 – Success

RDRAND	int get_rdrand16u(uint16_t *rng_val, unsigned int retry_count)	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)	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)	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)	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)	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, unsigned int retry_count)	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 retry_count - Number of retry attempts Return type *int*: Success or failure status of function call
RDSEED	int get_rdseed16u(uint16_t* rng_val, unsigned int retry_count)	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)	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)	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)	Returns 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(uint32_t *rng_arr, unsigned int N, unsigned int retry_count)	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, unsigned int retry_count)	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 retry attempts Return type *int*: Success or failure status of function call

Table 1. AMD Secure RNG APIs

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. Below code snippet 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 u
sing 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 of Cryptography
- Authentication protocols
- Internet Gambling
- Encryption
- Seeding software based pseudo-random number generators (PRNG)

AMD's latest microprocessor architecture, codenamed "Zen", has a strong focus on the cryptography domain with dedicated hardware block, Cryptographic co-processor in the AMD Secure Processor. AMD Secure RNG library provides a suite of APIs which developers can easily make use of in their applications.

Secure RNG performance

Each invocation of RDRAND and RDSEED instruction reads a value from Cryptographic Co-processor block. The read operation, being an Memory-mapped I/O(MMIO)[4] access, is relatively slow compared to other software based PRNGs. However, in cryptography applications which are not very latency sensitive, this may not be much of an issue. In specific cases, where they are latency sensitive, one recommendation is to use a hybrid approach, where a PRNG implementation can be seeded using a value from the hardware generated random value/seed. This would serve the purpose of adding certain degree of security as well as better performance.

Conclusion

Hardware based random number generators provide more secure and robust random values than software implemented generators. AMD Family 17h processor platform includes Random Number Generator design in its CCP 5.0 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 cryptography key generation and many others can benefit from the library and get access to high quality random numbers.

REFERENCES

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

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

[3] AMD Random Number Generator Whitepaper by David Kaplan, AMD http://support.amd.com/TechDocs/amd-random-number-generator.pdf

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

DISCLAIMER

The information contained herein is for informational purposes only, and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale.