SecurityDynamics

RSA Data Security
A Security Dynamics Company

Using RSA BSAFE® Crypto-C with the Intel® Random Number Generator

An RSA Data Security White Paper

RSA Data Security, Inc.
2955 Campus Drive, Suite 400
San Mateo, CA 94403
Phone: 650-295-7600
Fax: 650-295-7700
http://www.rsa.com

Security Dynamics Technologies, Inc.
20 Crosby Drive
Bedford, MA 01730
Phone: 781-687-7800
Fax: 781-687-7010
http://www.securitydynamics.com

Introduction

Good random number generation is vital to cryptographic security. Operations such as RSA keypair generation, symmetric key generation, random challenges, and initialization vectors all require good random numbers in order to be secure. These random numbers are typically provided by special algorithms called pseudo random number generators (PRNGs) in software or specialized hardware random number generators (RNGs). PRNGs are software algorithms that take a random seed and generate streams of random bytes.

The non-cryptographic random-number generators built into most compilers are highly insecure and should not be used for any security application[1]. However, the hardware-based Intel Random Number Generator included in the Intel 8XX series of chipsets provides high quality random numbers suitable for use in security applications. These chipsets are shipping in computer systems beginning in mid-1999.

The RSA BSAFE® Crypto-C software security component includes PRNGs that are designed to ensure good algorithmic properties. To produce random number output that cannot be predicted, the BSAFE Crypto-C PRNGs must be "seeded." A *seed* is unpredictable input, generated by hardware or by software, that is used to set the initial state of the otherwise predictable algorithm of a PRNG. For security reasons, it is crucial that a PRNG is seeded with unpredictable random input. This seed input can be derived from a number of sources including software and hardware[2].

For seeding the BSAFE Crypto-C PRNG from a hardware random number generator, the Intel Random Number Generator (RNG) is a good option that enables your application to get the high-quality, high-entropy bits that are needed. Once seeded, the software random number generator is used to produce cryptographic keys and challenges that, in turn, can protect vast quantities of data. Therefore, you will wish to use a software PRNG, seeded with random values from the Intel RNG, if you need to demonstrate compliance with certain standards such as the US Government's FIPS, ensure security for your applications, and create the most efficient and secure implementations possible.

Property	Hardware RNG	PRNG
Fast "seeding," or initial output	X	
True randomness	X	
No user input required	X	
High throughput of random data		X
Continued operation if HRNG fails or is unavailable		X

Table 1: Hardware random numbers vs. software pseudo-random numbers

Scope of this Document

This paper details how applications built with RSA security components can access the Intel RNG to seed a RSA BSAFE Crypto-C software random number generator (PRNG). Existing BSAFE Crypto-C applications from RSA Data Security's over 500 licensees can be easily modified to leverage the Intel hardware security with four easy steps. New applications built with BSAFE Crypto-C can also easily leverage the Intel hardware RNG to help secure their applications.

Audience

This document is for application programmers who are familiar with BSAFE Crypto-C and wish to benefit from Intel's hardware-based security features in a BSAFE Crypto-C application.

The Intel Random Number Generator

The Intel Random Number Generator is dedicated hardware that harnesses system thermal noise to generate random and indeterministic values. The generator is free-running, accumulating random bits of data until a 32-bit buffer is filled. In addition, the bits supplied to the application have been mixed with a SHA1 hash function for added security under extreme conditions of voltage and temperature. Future versions of Intel hardware will include additional hardware security building blocks that increase software security.

Hardware Availability

Since the Intel RNG is only introduced beginning in the middle of 1999, it is possible that hardware RNG may not be available. Therefore, application developers need to plan to encounter older systems without hardware random number generators. In this case or if the hardware RNG fails, the BSAFE Crypto-C interface to the Intel RNG will return an error condition that the hardware RNG is unavailable and the application must make an important security decision about where to get this seed. For example, an application that does not require extremely high security may wish to inform the user and use software means such as user input to seed the random number generator in BSAFE Crypto-C. Currently, the majority of applications use software methods to seed random number generators. However, extremely high-security applications might want to inform the user and exit should hardware not be available.

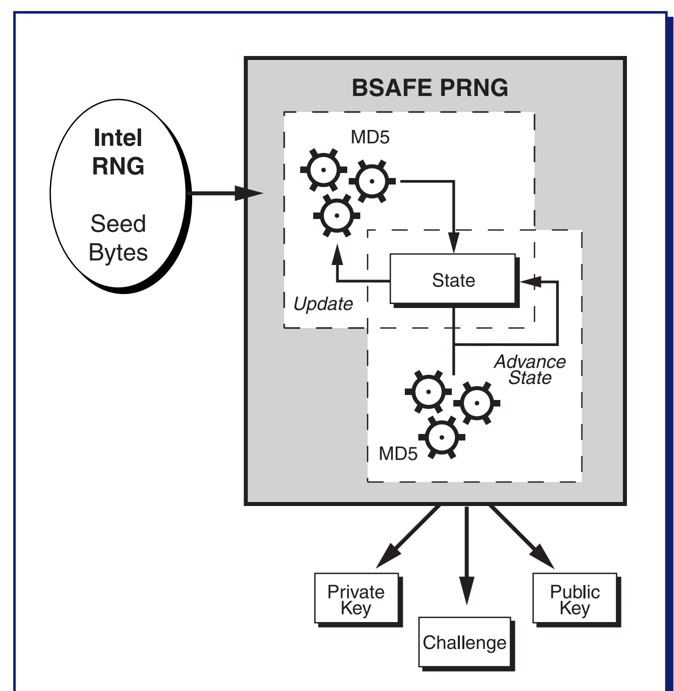

Figure 1: Interaction of the Intel RNG with the BSAFE PRNG

The BSAFE Crypto-C Pseudo-Random Number Generators (PRNGs)

BSAFE Crypto-C provides several cryptographically strong pseudo-random number generators that can be seeded via the Intel RNG and used to generate random numbers. The PRNGs in Crypto-C satisfy mathematical tests that are considered to indicate randomness. A pseudo-random number generator can be used in conjunction with the Intel RNG by using the Intel RNG to provide a *seed*. Once a PRNG has been seeded, it produces output up to ten thousand times faster than a hardware random number generator. For most applications, using a PRNG that has been randomly seeded by the Intel Random Number Generator is the best solution to provide the level of security needed since it will be faster, will avoid any potential problems due to hardware failure, and also operate on platforms other than those based on Intel hardware.

Generating Random Numbers

To generate random numbers, do the following:

1. Use the Intel Random Number Generator to generate a random seed. Because the software PRNG will be used to generate multiple random numbers for any given application, you should use a seed that is as large as the largest pseudo-random value that you plan to generate. In most applications you will need to generate multiple random numbers with the same PRNG. For example, if you will be generating five random numbers, and the largest number you will be generating is 1024 bits, you should use at least a 1024-bit seed to ensure security.

2. Seed a pseudo-random number generator with the seed that you generated in the first step. Once you have gotten a seed, you can use the PRNG to generate your random numbers. If you are already using a BSAFE Crypto-C PRNG in your applications, making the change to use the Intel Random Number Generator is easy. All you have to do is gather the seed as in 1 above, then make some minor changes to your existing implementation so that it can use the seed supplied by the Intel RNG.

Obtaining a Random Seed from Hardware

First, use the Intel Random Number Generator to acquire a random seed.

Step 0: Creating the Session Chooser

The first step in creating an application that can access the Intel Random Number Generator is to create the session chooser that associates Intel's hardware method, `HW_INTEL_RANDOM`, with Crypto-C's generic method for hardware random number generation, `AM_HW_RANDOM`. First set up your software and hardware choosers, then call `B_CreateSessionChooser`. This call will combine the elements of the software chooser with those in the hardware chooser, associating Intel's hardware method (`HW_INTEL_RANDOM`) with `AM_HW_RANDOM`, so that when `AM_HW_RANDOM` is called, Crypto-C turns to the hardware. For more information see the section on "The Session Chooser" in the *BSAFE Crypto-C User's Manual*.

```
B_ALGORITHM_METHOD *SOFTWARE_CHOOSER[] = {
   &AM_HW_RANDOM,
   (B_ALGORITHM_METHOD *)NULL_PTR
 };
  HW_TABLE_ENTRY *HARDWARE_CHOOSER[] = {
   &HW_INTEL_RANDOM,
    (HW_TABLE_ENTRY *)NULL_PTR
 };
  if ((status = B_CreateSessionChooser
       (SOFTWARE_CHOOSER, &CHOOSER, (POINTER *)HARDWARE_CHOOSER,
       (ITEM *)NULL_PTR, NULL_PTR, &oemTagList)) != 0)
   break;
```

Step 1: Creating an Algorithm Object

The next task is to create the algorithm object. This object will control the random byte generation. Creating the object only allocates the memory needed for the process. It does not initialize the object for random number generation.

```
if ((status = B_CreateAlgorithmObject (&randomAlgorithm)) != 0)
   break;
```

Step 2: Setting the Algorithm Object

Set the algorithm info. We will specify `AI_HW_Random`, which will point to the hardware method that was associated to `AM_HW_RANDOM` in the call to `B_CreateSessionChooser`.

```
if ((status = B_SetAlgorithmInfo
        (randomAlgorithm, AI_HW_Random, NULL_PTR)) != 0)
    break;
```

Step 3: Initialize the Random Object

Initialize ***randomAlgorithm*** to generate random bytes. Here we pass the ***CHOOSER*** that was created via the call to `B_CreateSessionChooser` above. This chooser contains pointers to the hardware method that was associated with `AM_HW_RANDOM`.

```
if ((status = B_RandomInit
        (randomAlgorithm, CHOOSER, (A_SURRENDER_CTX *)NULL_PTR)) != 0)
    break;
```

Step 4: Update the Random Object

Step 4 is not needed for random number seeding in hardware.

Step 5: Generate Random Bytes

Generate the random bytes for the seed. In this example, you will have the Crypto-C SDK generate 96 random bytes, storing the data in *seedBytes*. The last parameter is a surrender context. In this case, generating 96 random bytes should be very quick, so you can pass in a properly cast `NULL_PTR`.

```
if ((status = B_GenerateRandomBytes
      (randomAlgorithm, seedBytes, seedMaxLength,
        (A_SURRENDER_CTX *)NULL_PTR)) != 0)
    break;
```

Step 6: Destroy

Step 6a: Destroy the Algorithm Object

Destroy the algorithm object. This step will free any allocated memory used by `randomAlgorithm`. The memory is overwritten with zeros before it is deallocated, so that any potentially sensitive information is not left in memory.

```
B_DestroyAlgorithmObject (&randomAlgorithm);
```

Step 6b: Free the Session Chooser

Free the session chooser. It is important to free the session chooser, so that any handles to hardware and allocated memory are released.

```
if ((status = B_FreeSessionChooser (&CHOOSER, &oemTagList)) != 0)
   break;
```

Generating Random Numbers with BSAFE Crypto-C

Once you have a random seed, you can generate pseudo-random numbers in software with BSAFE Crypto-C. Once the seed has been passed to the software algorithm info type, this is similar to any Crypto-C PRNG implementation. The only difference is the fast, truly random seed operation. For this example, you will use Crypto-C's SHA1 PRNG to generate random numbers.

The example in this section is almost identical to the example in the *Crypto-C User's Manual*, "Generating Random Numbers." Steps 1, 2, 3, and 6 are identical; the only difference is in the seeding of the PRNG in Step 4 and the random number generation in Step 5.

Note: For this software call, you do not need to create a special session chooser. A standard Crypto-C software chooser is sufficient.

Step 1: Creating an Algorithm Object

As before, you need to start by creating an algorithm object. This is identical to the software implementation

```
if ((status = B_CreateAlgorithmObject (&randomAlgorithm)) != 0)
   break;
```

Step 2: Setting the algorithm object

To set the random algorithm object to use Crypto-C's SHA1 random number generator, you need to supply the appropriate algorithm info type. For SHA1, this is `AI_X962Random_V0`. Again, this is identical to a software implementation.

Note: This algorithm info type is named after the standard where the pseudo-random number generator is defined, Because SHA1 is considered one of the most secure implementations for creating pseudo-random numbers, there are a number of SHA1 random number generators in the literature. All of them use SHA1, but may differ in certain implementation details. Therefore, the AI is named after the standard for clarity and precision.

```
if ((status = B_SetAlgorithmInfo
      (randomAlgorithm, AI_X962Random_V0, NULL_PTR)) != 0)
   break;
```

Step 3: Init

Initialize the random algorithm. You must pass the algorithm object, the algorithm chooser, and a surrender context. As mentioned before, the algorithm chooser does not need to be a session chooser; a simple software chooser will suffice, so this call is also identical to a software implementation.

```
B_ALGORITHM_METHOD *RANDOM_CHOOSER[] = {
   &AM_SHA_RANDOM,
   (B_ALGORITHM_METHOD *)NULL_PTR
 };
  if ((status = B_RandomInit
        (randomAlgorithm, RANDOM_CHOOSER,
         (A_SURRENDER_CTX *)NULL_PTR)) != 0)
     break;
```

Step 4: Seeding the Random Object

In this step, you will seed the random object using the seed generated by the Intel RNG. If the RNG cannot be found, or fails during processing, you can ask the user to enter a seed value. Most applications today use system entropy, such as mouse movements and keyboard input to seed software PRNGs. Extremely high security applications that are unable to use the Intel RNG may wish to inform the user and exit. Other applications may wish to use software seeding instead.

First, acquire the random seed:

```
randomSeedLen = 96;
  randomSeed = (unsigned char *)T_malloc (randomSeedLen);
  GenerateSeed (randomSeed, randomSeedLen);
```

Once you have the random seed and its length, pass both into `B_RandomUpdate`. This call would be identical in a software implementation:

```
if ((status = B_RandomUpdate
      (randomAlgorithm, randomSeed, randomSeedLen,
       (A_SURRENDER_CTX *)NULL_PTR)) != 0)
   break;
```

Step 5: Generating Random Numbers

First, before calling `B_GenerateRandomBytes`, prepare a buffer for receiving the random bytes. This is a little different than the software implementation.

```
RandomByteBuffer = T_malloc (BYTES_TO_GENERATE);
 if ((status = (randomByteBuffer == NULL_PTR)) != 0)
       break;
  T_memset (randomByteBuffer, 0, BYTES_TO_GENERATE);
```

Now you can generate the random bytes. Since generating 64 bytes is quick, you can use a `NULL_PTR` for the surrender context:

```
if ((status = B_GenerateRandomBytes
      (randomAlgorithm, randomByteBuffer, BYTES_TO_GENERATE,
       (A_SURRENDER_CTX *)NULL_PTR)) != 0)
   break;
```

Step 6: Destroy

Remember to destroy all objects when you are done with them, and free all memory. Again, this is identical to the software implementation:

```
B_DestroyAlgorithmObject (&randomAlgorithm);
 T_memset (randomSeed, 0, randomSeedLen);
 T_free (randomSeed);
 T_free (randomByteBuffer);
```

Conclusion

Random number generation and seeding using the Intel RNG allows BSAFE Crypto-C to generate better random numbers. This improves the security provided to existing and new products built with BSAFE Crypto-C. Applications can easily harness the security provided by the Intel hardware security features by only changing a few lines of code.

Notes:

[1] If a third party is able to reproduce the random numbers used in an application, much of the sensitive data may be at risk. For example, when generating keys, a random number generator is used. If it is possible to find the random numbers used to generate the keys, it is possible to derive the keys used in cryptographic operation and expose sensitive information.

[2] In situations where hardware random number generation is not possible, RSA Data Security provides guidelines on how to collect random seeds. These guidelines, and other technical notes, are available on the Web at the RSA Labs' website at http://www.rsa.com/rsalabs/bulletins.html

Copyright 1999, RSA Data Security, Inc. All rights reserved.
BSAFE is a registered trademark of RSA Data Security, Inc.
All other company and product names are the property of their respective owners.