Random Number Generation in Java

Ines Duits

25 October 2015

Abstract

Random number generators are used in a lot of software, and are especially needed for cryptography purposes. Random numbers generators needs to be unpredictable, not bias and uncorrelated. There are two different kind of random number generators (RNG): true RNG which use quantum mechanics, noise from outer space or radioactive decay as the source of randomness. And pseudo RNG, which use algorithms and an initial value, the seed, to generate random numbers. Pseudo random numbers generators are deterministic, given the same seed, they will generate the same output. In the programming language Java, there are two different classes to generate random numbers which both uses pseude RNG. The *Random* class generates random numbers with a simple linear formula, which make it easy to predict, this works fine for simulations and games, but not for cryptographically purposes. The *SecureRandom* class uses the *SHA1* algorithm, to generate random numbers. We see that today it has becomes easier to attack this algorithm and therefore we may want to follow the advice to use a newer version of *SHA* instead of *SHA1* for the generation of numbers in the *SecureRandom* class.

1 Introduction

Technology is changing fast and so there is the need for a good way to secure information. We want everything to be securely encrypted. There are a lot of algorithms like RSA, tripleDES, AES, WPA [18], which are used to keep our information safe and away from unwanted eyes. These algorithms need random numbers to work more secure (read: better) [13]. But how do computers generate random numbers? Computers need to generate random numbers all the times, in simulations, games, other software and for security reasons. In the program languages Java and C♯ we have a random number

class and a secure random number class. But why is there a need for two different classes? Is the normal random number class not random enough? Which random number generator do we want to use in cryptography? Probably the secure random number generator, but why?
In this paper we will look at the classes *Random* and *SecureRandom* in the programming language Java to see if there are secure random. First we will look at what a random number is and when we may call it truly random. How can we generate a random number which can be used in cryptography? When we have a better idea about random numbers and random number generators, we look at an example from Java. We will explain how the *Random* and *SecureRandom* classes generate random numbers. Then we may say if they are secure or not.

2 Random

We start simple: what is random? The online dictionary of Cambridge[4] gives us the following definition *"Happening, done, or chosen by chance rather than according to a plan"*. When throwing a dice, we expect to get a random number between 1 and 6. And there is a 16 chance of to get any number on the dice. In fact, there are some theories[17] that claim that throwing dices are not truly random and pure change, but these theories are beyond the scope of this paper. For us throwing a dice is a random event, and throwing a dice multiple times gives us a random number generator. But computers can not throw a dice (actually they can, with a robot arm, but that is not what is meant here). Computers need some software based random number generator. From now one when we speak of a number generator (NG) we mean a binairy number generator, unless stated otherwise. What makes a throw of a dice, or any other number generator random?

Statistics

We start to state a number generator is random when it is *unpredictable*[6] whether the next bit is going to be one or zero. But unpredictable does not automatically mean that a number generator is random [15]. We can say that the generation process needs to be *not baised*, there has to be a change of 1/2 to get a one, and 1/2 to get a zero. And the probability of generation a zero or a one may not depend on the previous bit(s), they may *not be correlated*. There are a few statistical tests which a number generator needs to pass before we *may* call it random [6]. Note that it is impossible to give a mathematical proof to show a number generator is random [13].

The statistical tests state that a number generator is not rejected of being a random number generator. We only give a brief explanation of the tests, for more detail check The Handbook of Applied Cryptography[13] and Crocker, Cybercash and Schiller paper on Randomness Recommendations for Security [6], which were used to create the following list of tests.

Frequency test determines of the number of one's (n_0) and zero's (n_1) in a sequence are approximately the same. The generated values of the PRNG needs to be *distributed uniform*. The amount of generated zero and one bits needs to be almost the same.

Serial test determines of the number of 00, 01, 10 and 11 are approximately the same.

Poker test determines of a subsequence, from the random sequence, of length m does not appear more often that is expected from a random sequence.

Runs test determines that there are not more unanimous subsequence as expected from a random sequence.

Autocorrelation test determines of there is correlation between the sequence and shifted version of it. We now have a general idea about which properties make a number generator random. But how we transform this idea into actual random number generators will be explained in the next section.

3 Random number generators

We now know what a random number is, but how do we transform that idea into an actual random number generator? In literature there are actually two different kind of random number generators: true random number generators, described in section 3.1, and pseudo random number generators described in section 3.2. The introduction to random number generations in this section is written mostly by information found in The Handbook of Applied Cryptography[13] and Crocker, Cybercash and Schiller paper on Randomness Recommendations for Security [6], for more detail please read their work.

3.1 True random number generators

A true random number generator requires a natural source of randomness. A simple example is the dice, if we do not cheat with it ect. But for a computer we another source of randomness. We could use thermal noise from outer space, hardware which use radioactive decay [5] [13] and quantum processes [5] [7]. The noise data needs to be interpreter and de-skewed to make it pass the statistical tests as stated in Section 2.
And the interpretation of the data will be done by some sort of software algorithm which is know by the enemy, by Kerckhoffs Principe [18]. How easy is it for Oscar to get the same noise and use the algorithm to get the random numbers? There are a lot of things that needs to be make secure if we used true random number generations. And you can even discuss if it is truly random, because you need to interpreter the data before you can actually use it. And because, who says the noise from outer space is random? But all of this is out of the scope of the paper! In this paper we focus on *pseudo* random number generators.

3.2 Pseudo random number generators

Pseudo Random Number Generators (PRNG) will generate random numbers by the use of mathematical algorithms. The generated numbers look statistically random but that does not simply imply that they are truly random. Most PRNG use a seed, a value from which it will starts to generate numbers [6]. Depending on the seed, the PRNG generates a random sequence of numbers. Pseudo random number generators are deterministic, given the same seed, the algorithm produce the same sequence of random numbers[13]. A simple example of a pseudo random number generator is a linear congruential generator[13]. With the formula

$$x_n = (ax_{n-1} + b)mod(m)$$

In which x_0 is the seed and a, b and m parameters, there will be generated something that may look random. It may pass al the statistical test, stated in Section 2 but will give a predictable sequence, especially when you know the begin state, the seed.

3.3 Seed

The seed in a pseudo random number algorithm is the start value which is used for the algorithm. It implies how random a PRNG will become, or how predictable [6]. It is, in fact, like the keys used in security algorithms, once

you know the key, you can decrypt the message: once you know the seed, you can predict the random output.
The source of a seed can be data read from your computer, like the current time or date, serial numbers, mouse movement, key strikes, packets arrival time or the sound or camera input from the computer. But that may not be good random seeds. First it is possible to pretend the same "time" on a machine just to get the same PRNG seed, because of the deterministic property of the algorithm. The same holds for all the other examples: they can be observerd and/or faked [6] [5]. To generate a better seed we may consider a combination of uncorrelated data from your computer. If we combine these to a seed an use this as an input to a strong mixing function, we get less predictable random number generators [6]. Secondly this seeds may not meet the requirements stated in Section 2. But therefore we can use a so called mix function[13]. Mixing functions are one time functions which mix any given input into an output which shows nothing of the initialized input. The PRNG algorithm may even include a mixing function. There are other ways to get or generate a seed, but we do not discuss them here.

4 PRNG in Programming languages

Most programming languages have two random classes: the normal and the secure class. The normal random classes generate random looking numbers which may pass the statistical tests given in section 2 but which are easy to predict [12] [8]. For instance the Microsoft .NET System.Random class [1], used in C#, generates random numbers using an algorithm that selects numbers from a finite set of numbers and thereby is not random, this is clearly explained in the documentation[1] remark. The same holds for the Random API from Java [8], which uses the linear congruential generator formula, shown in section 3.2, to generate random numbers.
These random classes may be used in simulations or a game, but definitely not in cryptography! Therefore programming languages added special secure random classes, in C# *Security.Cryptography* and in Java *SecureRandom*, which can be used in cryptography. A disadvantage of these classes is that they are slower[12] and most of the time need to generate more bits, for a sufficient entropy, before they can be used. In the following sections we will look at how the *SecureRandom* class works in the Java programming language.

4.1 SecureRandom in Java

Secure random numbers in Java are generated with the SecureRandom class. In most (Windows) applications the default value, *"SecureRandom("SHA1PRNG")"*, is used [16] [8], which uses the SHA1 algorithm. Note that this is the default value on Windows. Linux uses the default *"SecureRandom("NativePRNG")"*, which we discuss in section 4.5. In the next section we will look at how the SHA1 algorithm works.

4.2 Secure Hash Algorithm 1 (SHA1)

We will now look at generalized description on how the SHA1 algorithm works. For a more detailed explanation look at P. Jones' paper on US Secure Hash Algorithm 1 (SHA1) [10] and the video by Prof. Cristof Paar [14] which were used to create this description.
The input for the SHA1 algorithm is a binary string. The given input will be padded, filled up with bits to a total length of 512 bits; one 1bit followed by only zeros bits. The padded input, we call x, gets split into 80 words, W_j, with a length of 32 bits. For the first 16 words we use the direct value of x, W_0 will have the value of the first 32 bits of x, W_1, the second 32 bits etcetera until W_{15} has the last 32 bits of x. For W_j with $16 \leq j \leq 79$ we will use the formula $W_j = W_{j-16} \oplus W_{j-14} \oplus W_{j-8} \oplus W_{j-3}$.

The words will now be processed through 80 rounds, one round for each word W_j. These 80 rounds are split in four stages, stage t=0 with W_j where $0 \leq j < 19$, stage t=1 with W_j where $20 \leq j < 39$, stage t=2 with W_j where $40 \leq j < 59$ and stage t=3 with W_j where $60 \leq j < 79$. We initialize this process with five fixed words: A, B, C, D and E, all with a length of 32 bits. These five words are used to be add (with XOR) to the output value after the 4 stages, or 80 rounds. The algorithm is schematically displayed in Figure 1.
In each round we apply a function F_t on B, C and D, XOR that with E, getting E'. Rotate A five bits, XOR that with E' getting E". XOR W_j with E" getting E"', and XOR E"' with K_t, K_t is a constant with a length of 32 bits, depending on the stage. The outcome of that we use as the A in the next round. The starting values of A, C and D will become the B, D and E in the next round and the starting value of B will be shifted 30 bits to the left and become the new C. This process is displayed in figure 2.
The function F_t and the constant K_t depends on the stage. The exact values of this constants and the initialize values A, B, C, D and E can be found in the Appendix A at the end of this paper. The end values of A, B, C, D and

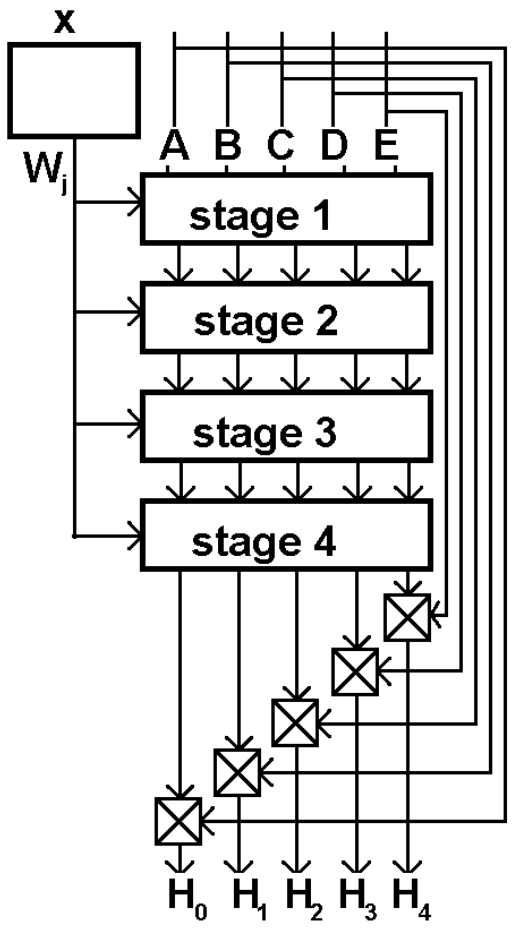

Figure 1: The algorithm with 4 stages

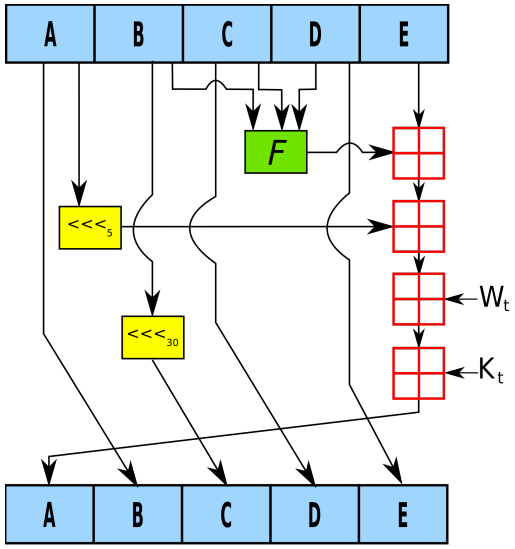

Figure 2: The algorithm for round j. Source: `https://en.wikipedia.org/wiki/SHA-1`

E, which are called H0, H1, H2, H3 and H4 in figure 1, are combined and become the output.

4.3 Safe

Regardless of how safe the seed may be, we may want to consider how secure the SHA1 algorithm is. There is a lot of discussion about this. The SHA1 algorithm uses a random seed and generate a 160 bit value from it, which is used for the secure random seed. There are 2^{160} different outcomes of this algorithm, and there may be some collision: two different seeds which give the same SHA1 value. The basic techniques to attack hash algorithms are described in Gerard Tel's book on Crytography [18]. In 2005 Bruce Schneier and John Kelsey decribed way to attack SHA1 in $O(2^{106})$, in their paper *Second Preimages on n-bit Hash Functions for Much Less than* 2^n *Work* [11]. In 2013 Marc Stevens explained how it can be done in $O(2^{61})$ [2]. Which is, in the current value of number of operations, in the danger zone[2]. Thereby we need to say that the SHA1 algorithm was, but no longer is, a standard advised hash function.
The security may depend on the seed, the initial input. But regardless of the seed, the danger as state above holds for the SHA1 with a unknown seed.

4.4 Seed

We understand how the SHA1 hash function works. But to actually generate a random number we need some seed. There are two default ways in which the algorithm gets it random input, called entropy [3]. On a Windows system the algorithm gets it directly from *CryptGenRandom* and on a Linux system from the file */dev/urandom*. Both work with a lot of random user input, like mouse movement, file directories, memory uses, which in some ware are combined to a seed. On a system where the files */dev/random* and */dev/urandom* are available (properly a Linux system), these files can be both used as a seed in the SHA1 algorithm and as the random number input, which we describe in the next secion.

4.5 NativePRNG

The *NativePRNG* implementation of Java is another way to generate random numbers[8], and it is the default value for the *SecureRandom* class in Java on Linux systems. It reads directly from */dev/urandom* or */dev/random*, if those files are available, otherwise *NativePRNG* cannot be used. The files contains random bits which they got by multiple unpredictable source from the operation system, like keyboard events, mouse movements and memory usage, which we discussed in section 3.3. The random inputs are processed with i.a. the SHA1 mixing function, see section 3.3 and section 4.2, to fill the entropy pool[9]. Therefore both files use a mixing function before feeding the entropy pool.
The difference between the two files is that *random* will stop when the entropy pool does not have enough data, it will block and is not useable until there is new data. This means that the *random* file still has random bits available but it blocks because of the small amount of bits, because it may not be random enough. This cases errors when we ask more random bits than the entropy pool has. The *urandom* file in the other hand will not block when the entropy pool of *urandom* runs out of data [5] [9].
There is a bit of a discussion in the Linux community about which file to use. Because of blocking of the *random* file, some people prefer the *urandom* file, but others think it is not safe enough. Read Thomas Hühn's post *"Myths about /dev/urandom"* if you are interested in this discussion [9].

5 Conclusion

A sequence of random numbers needs to be uncorrelated, has an uniform distribution and unpredictable (not bias). If that holds for a certain generator

we may call it a true or pseudo random number generator. It is impossible to mathematically proof that a sequence or random number generator is random! And there is a lot of discussion possible about whether something is truly random or not.
Apart from that discussion about what random is, we looked at *Random* classes, from C# and Java, which were very predictable and easy to attack. Using it in simulations and game is fine but we would advise against using these in any software which needs to be safe and secure or where you do not want your user to predict the outcome. The *SecureRandom* in Java is much more secure, because it uses the SHA1 algorithm in combination with a random seed combined from random user and OS input. But we saw that SHA1 becomes easier to attack, and with faster computers every year this will become more easy.
We may conclude that the *SecureRandom* class, using the SHA1 algorithm is better and more secure. And you definitely want to use the *SecureRandom* class when you want your random numbers to be more secure and unpredictable. We saw that we may even need a better *SecureRandom* class for more serious secure purpose, but for a normal game, simulation or a simple encryption system the *SecureRandom* class will be secure enough!

References

[1] Documentation random class .net framework 4.6 and 4.5. URL `https://msdn.microsoft.com/en-us/library/system.random.aspx`.

[2] New attacks on sha-1: optimal joint local collision analysis, 2013.

[3] Securerandom implementation (sun.security.provider.securerandom – sha1prng), 2014. URL `https://www.cigital.com/blog/securerandom-implementation/`.

[4] Cambridge online dictornary, 2015. URL `http://dictionary.cambridge.org/dictionary/english/random`.

[5] R. Connolly. Entropy and random number generators in linux, 2007. URL `http://www.linuxfromscratch.org/hints/downloads/files/entropy.txt`.

[6] S. Crocker, Cybercash, and J. Schiller. Randomness recommendations for security. 1994. URL `http://www.ietf.org/rfc/rfc1750.txt`.

[7] J. Emerson, Y. S. Weinstein, M. Saraceno, S. Lloyd, and D. G. Cory. Pseudo-random unitary operators for quantum information processing. *SCIENCE*, 302:2098–2100, 2003. URL http://www.researchgate.net/profile/Yaakov_Weinstein/publication/8951276_Pseudo-random_unitary_operators_for_quantum_information_processing/links/09e4150ff080801dcc000000.pdf.

[8] P. Garg. Secure random number generation in java, 2011. URL http://resources.infosecinstitute.com/random-number-generation-java/.

[9] T. Hühn. Myths about /dev/urandom, 2014. URL http://www.2uo.de/myths-about-urandom/#low-entropy.

[10] P. Jones. Us secure hash algorithm 1 (sha1). *Cisco Systems*, Network Working Group, 2001.

[11] J. Kelsey and B. Schneier. Second preimages on n-bit hash functions for much less than 2^n work. 2005. URL http://eprint.iacr.org/2004/304.pdf.

[12] B. Klopfer. How to generate better random numbers in c sharp .net, 2012. URL http://thinketg.com/how-to-generate-better-random-numbers-in-c-net-2/.

[13] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *The Handbook of Applied Cryptography*. 1996. URL http://labit501.upct.es/~fburrull/docencia/SeguridadEnRedes/teoria/bibliography/HandbookOfAppliedCryptography_AMenezes.pdf.

[14] P. C. Paar. Sha-1 hash function, 2011. URL https://www.youtube.com/watch?v=5q8q4PhN0cw.

[15] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10), 1988. URL http://www.firstpr.com.au/dsp/rand31/p1192-park.pdf.

[16] A. Sethi. Proper use of java securerandom, 2009. URL https://www.cigital.com/blog/proper-use-of-javas-securerandom/.

[17] B. P. Stein. Dice rolls are not completely random, 2012. URL https://www.insidescience.org/blog/2012/09/12/dice-rolls-are-not-completely-random.

[18] G. Tel. *Cryptografie, beveiliging van de digitale maatschappij*. 2006.

APPENDIX A. Constant values in SHA1

The constant and initial values used in the SHA1 are given here. The values were taken from at P. Jones' paper on US Secure Hash Algorithm 1 (SHA1) [10] . The initialized values, A, B, C, D and E of the algorithm are

$$A = 67452301$$

$$B = EFCDAB89$$

$$C = 98BADCFE$$

$$D = 10325476$$

$$E = C3D2E1F0$$

all given in hexadecimal.
The stage constants K_T are

$$K(t) = 5A827999 \qquad (0 \leq t \leq 19)$$

$$K(t) = 6ED9EBA1 \qquad (20 \leq t \leq 39)$$

$$K(t) = 8F1BBCDC \qquad (40 \leq t \leq 59)$$

$$K(t) = CA62C1D6 \qquad (60 \leq t \leq 79).$$

Also given in hexadecimal.
And the functions F_t, depending on the stage are:

$$f(t; B, C, D) = (B \wedge C) \vee ((\neg B) \wedge D) \qquad (0 \leq t \leq 19)$$

$$f(t; B, C, D) = B \oplus C \oplus D \qquad (20 \leq t \leq 39)$$

$$f(t; B, C, D) = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) \qquad (40 \leq t \leq 59)$$

$$f(t; B, C, D) = B \oplus C \oplus D \qquad (60 \leq t \leq 79)$$

where $/oplus$ is the XOR operation.