IDC HERZLIYA | Efi Arazi School of Computer Science

The Interdisciplinary Center, Herzlia
Efi Arazi School of Computer Science

Pseudo Random Number Generators in Programming Languages

M.Sc dissertation

Submitted by **Aviv Sinai**

Under the supervision of Dr. Zvi Gutterman
(CloudShare, HUJI)

March, 2011.

Acknowledgments

First and foremost, I would like to thank my advisor, Dr. Zvi Gutterman, for the time and effort he put into helping me complete this work.
I would like to express my deepest gratitude to Asaf Rubin, a friend and co-worker. I'm grateful for his help and time spent assisting me in finishing this work.
I would like to also thank Danny Slutsky and Yaniv Meoded, who reviewed early drafts of this work.
Special thanks to Dr. Anat Bremler and the IDC M.Sc. CS program office, for their patience and help.
Finally, I want to thank my family who gave me the support I needed to invest precious time working to complete this work.

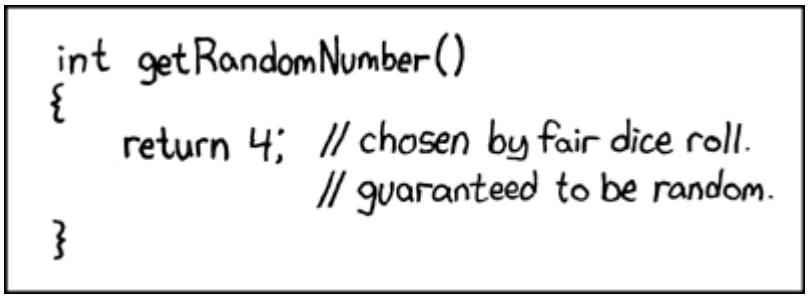
Abstract

Software developers frequently encounter the need to integrate random numbers in their systems and applications. Applications and systems that span from implementing a new security protocol to implementing a shuffling algorithm in an online poker game. Modern software languages come to their aid by providing them with a rich SDK that contains pseudo random number generation functions for the developer to use without the need to implement their own generators. These functions differ in cryptographic strength and underlying algorithms used.

In this thesis we research the implementations of random number generators in popular programming languages. We provide a complete and detailed analysis of the algorithms used, cryptographic strength and capabilities of these generators. Our analysis shows weaknesses in the generators implemented, including a bug in C#'s implementation of the additive feedback generator. In addition we provide a non-trivial attack on the session generation algorithm in PHP that relies on our analysis of PHP's generator.

Table of Contents

List of Figures


int getRandomNumber()
{
return 4; // chosen by fair dice roll.
// guaranteed to be random.
}

1 Introduction

1.1 Contributions

In this work we study the implementations, availability and security properties of Pseudo Random Number Generators (PRNGs) in popular programming languages. The algorithms used are methodically presented in concise pseudo-code format with a thorough analysis of their security parameters.

This work drives towards the goal that Knuth advised in [1]: *"… look at the subroutine library of each computer installation in your organization, and replace the random number generators by good ones. Try to avoid being too shocked at what you find."* Similar to these cautious lines from Knuth we see in [2] while discussing a *rand()* implementation: *"…Now our first, and perhaps most important, lesson in this chapter is: be very, very suspicious of a system-supplied rand() that resembles the one just described. If all scientific papers whose results are in doubt because of bad rand()s were to disappear from library shelves, there would be a gap on each shelf about as big as your fist."*

Most programming languages have several flavors of PRNG implementations for the programmer to choose from. The flavors differ in their security properties and sometimes also in their design and API.

This work has important practical and theoretical implications:

1. A PRNG is its own kind of cryptographic primitive, which all programming languages offer at least one implementation of. A better understanding of these implementations will make it easier to choose the correct implementation to use.
2. A PRNG is a single point of failure for many real-world cryptosystems. An attack on the PRNG can make the careful selection of good algorithms and protocols irrelevant.
3. Many systems use badly-designed PRNGs, or use them in ways that make various attacks easier than need be. Very little exists in the literature to help system designers choose and use these PRNGs wisely.
4. Most developers don't understand the difference between these primitives and tend to choose their PRNG flavor wrongfully.

In this work we concentrate on the analysis of 4 very popular programming languages (as claimed by the TIOBE Programming Community Index available in [3]): C (5), Java (Chapter 6), C# (Chapter 7) and PHP (Chapter 8). For each programming language, we survey the relevant information and papers that discuss the PRNG in that language. We then proceed to describe the exact implementation, design and configuration options that are available in this language.

The appropriate API documentation for each language usually served as a first step in the analysis. However, most of the documentation we encountered was extremely poor in its details of the PRNG, and/or consisted of various inaccuracies. In order to gain further insight to the exact implementations, we used static code analysis techniques. In addition, in some programming languages such as C#'s System.Security.Cryptography (Section 7.3), we were forced to reverse engineer the code using commodity tools such as the IDA-Pro disassembler [4].

After understanding the exact implementation and algorithms used we analyzed the security properties of the generators in a similar framework to which is described in [5] and [6].

For ease of reference for developers interested in the "bottom line", we also provide a complete summary table of the properties of each programming language and variant. The table can be viewed in Figure 35.

While analyzing the security of the implementations we found a bug in the PRNG implementation in C#'s *System.Random* generator (Section 7.2.3.1) – the bug causes the generator to not have

the maximal period length. Under certain relaxations we continued to analyze this bug and demonstrated concrete seeds that cause the generator to have an extremely short period of 2^{17} in its least significant bit. We further found a non-trivial attack on one of PHP's core PRNG generator.

We continued and showed an unpublished attack on the session generation mechanism in PHP (see 10 for details). The attack utilizes an attack we found in *lcg_value* (see section 8.2.3.1), which is one of the PRNG implementations that exist in PHP.

1.2 Structure and Outline

The rest of this work is structured as follows. In Chapter 2 we provide important background for this work, surveying applicable theoretical Pseudo Random Number Generators and explaining the properties of good PRNGs. In Chapter 3 we present the related work, which includes infamous attacks of PRNGs and analysis results of Operating System based generators work that will be referenced throughout this work. Chapter 4 comes to provide common context, language and the attack vectors used to analyze each programming language. Chapters 5, 6, 7 and 8 contain the actual analysis of the programming languages C, Java, C# and PHP, respectively. We present our conclusions and a summary table of our results in Chapter 9. Chapter 10 (Appendix A) contains our attack on the session ID generation in PHP and the rest of the Appendices are code extracted and/or used throughout the analysis. Due to the immense amount of code reviewed in each analysis we only present the code in the Appendices if reverse engineering (or other de-compilation methods) were needed to extract the code or if the code implementation didn't seem straight-forward to us.

2 Pseudo Random Number Generators

Real random number generators are hard to come by. These generators often require having specialized hardware and use physical sources such as thermal noise in electrical circuits or precise timing of Geiger counter clicks [7,8,9,10]. Due to this, most applications that require random bits use a cryptographic mechanism, called a Pseudo Random Number Generator (PRNG), to generate the (pseudo) random numbers.

In this chapter we discuss the importance of random numbers and provide some examples of the use of random numbers in popular applications. We continue to discuss the distinction between theory and practice in section 2.3 and finish with a survey of popular PRNGs in section 2.4.

2.1 The Importance of Random Numbers

Random numbers are prevalent in many computer science applications. These applications include network protocols design (e.g., TCP sequence number [11]), algorithmic research (e.g., random algorithms), various unique identifiers (e.g., UUID [12]) and security protocols (e.g., TLS [13]). Random numbers are considered a basic building block in almost every cryptographic scheme (e.g., RSA [14]).

Having a secure source of random numbers is a critical assumption of many protocol systems. There have been several high profile failures in random numbers generators that led to severe practical problems. Perhaps the most renounced one was in the Netscape implementation of SSLv2 (3.2) in 1996. For an overview of popular PRNG based attacks, please refer to 3.

SSL as an example of random numbers importance: The SSL protocol, which was originally developed by Netscape, is one of the basic building blocks that allow the Word Wide Web to function as we know it. E-commerce sites use SSL to secure online transactions; banks use SSL in order to secure sensitive communications of their clients and their servers; popular hosted solutions, such Google's Gmail (www.gmail.com), use SSL to secure their communications and many others. One can't even fathom the repercussions if security flaws in the implementation (or design) of the SSL protocol are to be found.

The security of SSL, as in many other security schemes, depends on the attacker not being able to predict the secret key of the scheme. Thus it is vital that this secret key would be derived from an unpredictable random source. Random numbers are used in several places in the SSL protocol. Random nonces are created during the *Handshake Protocol* and passed in the *Client Hello* and *Server Hello* messages. These nonces are important inputs to prevent replay attacks and are also used in deriving the future keys used for encryption. Most importantly, random numbers are used during the creation of the *pre master secret* that is sent by the client to the server during the *Key Exchange* phase in the handshake protocol (this actually serves as a secret key between the parties). Using weak random numbers in SSL would have the protocol crumble down.

An illustration of the SSL handshake protocol can be viewed in Figure 1 below.

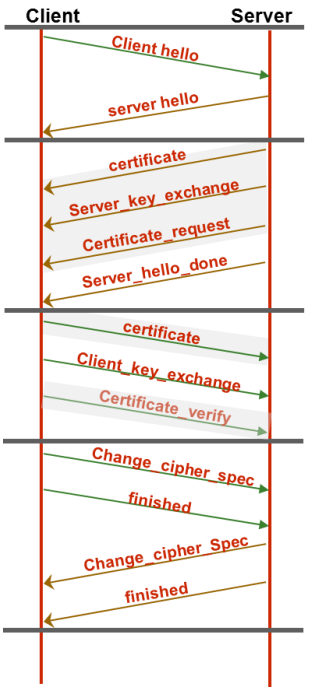


Figure 1 SSL Handshake Protocol Illustration[1]

2.2 What is a Good (Pseudo) Random Number Generator?

As noted earlier, obtaining randomness on a computer is not an easy task as a Turing machine is, by definition, deterministic. Generating real random numbers often involve having specialized hardware that is sensitive to physical bias, which needs post-processing tasks to remove this bias. Due to these, most applications use Pseudo Random Number Generators implemented in software when in need of random values.

We continue with definitions of PRNG by re-phrasing a bit some of the definitions available in the Handbook of Applied Cryptography [15].

Definition 1 (PRBG): *A Pseudo Random Bit Generator (or PRBG) is a deterministic algorithm which, given a truly random binary sequence of length k, outputs a binary sequence of length $l \gg k$ which "appears" to be random. The input to the PRBG is called the seed, while the output of the PRBG is called a pseudorandom bit sequence.* The initial random input of length k is referred to as the ***seed*** of the generator.

The purpose of PRNGs is to take a **small** real random sequence and expand it to a sequence of much larger length; in such a way that an adversary cannot efficiently distinguish between output sequences of the PRBG and truly random sequences of length l.

Definition 2 (SPRNG): *A PRNG whose output cannot be distinguished from a true random output by a polynomial time algorithm is a Secure PRNG (SPRNG).*

Random numbers are used in many applications; each implementation may have different requirements from its PRNG. Consider the need of having random numbers for simulations purposes - here the basic need for the random numbers is to have good (uniform) statistical properties. However, for instance, it is alright for the sequence to repeat itself from one simulation run to another. It is even **important** that the user can repeat simulations easy. This is, obviously, not the case with cryptographic systems.

[1] Illustration taken from Prof. Amir Herzberg "Introduction to secure communication and E-Commerce" lecture notes available at https://sites.google.com/site/amirherzberg/introductiontosecurecommunicationandcomm

In this thesis we are interested in the requirements of a PRNG from a **cryptographic** perspective. We continue to outline these requirements using common terminology coined in [5], which being a *SPRNG* is only one of them.

A PRNG must be secure against external and internal attacks. The attacker is assumed to know the code of the generator, and might have partial knowledge of the entropy used for refreshing the generator's state. Furthermore, the attacker might have the ability of compromising the internal state for a limited time. The **Security requirements** of a PRNG are**:**

- ***Pseudo-randomness***. The generator's output should seem random to an outside observer. This requirement is identical to the definition of an *SPRNG*. Even if the attacker is given all the output, the attacker can't be able to efficiently guess the next **bit** of the output.
- ***Forward security (or Backtracking Resistance)***. An adversary who learns the internal state of the generator at a specific time cannot learn anything about previous outputs of the generator. This requirement is easily met if the generator is a one way function.
- ***Backward security (or Prediction Resistance).*** An adversary who learns the state of the generator at a specific time does not learn anything about future outputs of the generator, provided that sufficient entropy is used to refresh the generator's state. The only means to use in order to meet this requirement is to periodically have the generator's state refreshed with new entropy.

2.3 Theory vs. Practice

A correct implementation of a PRNG is crucial. The design of the security protocol could be flawless; however an incorrect or weak implementation of the PRNG could cause the whole design to fail. Despite the fact that there are secure proven PRNGs for almost thirty years such as written in [16,17], many security protocols implementations are implemented with weak and vulnerable generators.

So the obvious question is why do we still suffer with so many inadequate generators used and implemented? (For some famous attacks and bugs on PRNGs the reader is encouraged to refer to chapter 3) In our opinion, the major reasons for this are:

1. **Performance** – as in the case of many theoretical concepts – when coming to implement an algorithm there are performance considerations and problems that are not always accounted in theory. Consider for example the generator by Blum, Blum and Shub [17], described in section 2.4.9. The generator's security is based on the difficulty to factor *semi-prime* numbers. However the algorithm itself is very CPU intensive, thus yielding in very few implementations actually using this generator.
2. **Attacks Due to Ecosystem** – many of the PRNGs described rely on sound mathematical basis and are usually even described as mathematical equations. However many attacks don't happen due to the fact that theoretical ground of the algorithm is shaky, but due to other practical considerations such as adhering to a specific API, meeting coding standards or design goals of the entire ecosystem.
3. **Level of Expertise** - most developers aren't versed in the field of Cryptography, nor are they aware of the potential delicacies in the field. Nevertheless, most developers would probably encounter during their career the need for generating random numbers. A poor choice of an API function might result in major security problems.

One of the main purposes of this work is to aid developers overcome the reason stated in Article 3 above in understanding the strength of each PRNG offered to them in their programming language of choice. This in turn would hopefully decrease the probability of choosing a PRNG implementation that is too weak for the application.

One of the most famous books that try to deal with the lack of expertise claimed above is the book ***Writing Secure Code***, published in 2002 by Microsoft's Michael Howard and David LeBlanc [18]. In this book the authors take the time to discuss the proper way of using the random generators available in Microsoft run-time libraries. Most importantly they suggest not using some of the weaker variants in cryptographic sensitive application. The reader is encouraged to read chapter 8 in the book for more details.

2.4 Popular PRNGs Review

In this section we review the theoretical principles, algorithms and properties of PRNGs that are mostly based on algebraic concepts, and are used as building blocks for the PRNGs implementations in various programming languages. We will use this review as reference when stating the theoretical PRNG of each implementation in applicable analysis sections.

2.4.1 Linear Congruential Generator (LCG)

Probably the most famous and popular PRNG implemented today; we found this type of generator in almost every programming language covered as one of the basic generators available. It is based on the scheme introduced by D. H. Lehmer in 1949 [19].

LCG is based on the following recurrence:

$$X_{n+1} = (aX_n + c) mod\ m, \qquad n \geq 0$$

Where *a*, *c* and *m* are constants, and X_0 is the seed.

Choice of a, c and m: in order to guarantee a **full period of $\boldsymbol{m}$** care must be taken when choosing these parameters. Knuth [1] has a detailed discussion of the properties that these parameters should meet. We summarize his recommendations here:

1. c and m should be relative primes.
2. $b = a - 1$ should be divisible by all prime factors of m.
3. b should be a multiple of 4 if m is a multiple of 4.

2.4.2 Multiplicative Congruential Generator (MRG/MCG/MLCG)

An MCG (Multiplicative Congruential Generator) is an LCG that has $c = 0$ in its recurrence. According to [1] this was actually Lehmer's original generation method, although he did mention $c \neq 0$ as a possibility.

The random number generation of this generator is slightly faster in this case; we note that this generator doesn't satisfy recommendation 1, thus it can't achieve the full period. Here we want X_n to be relatively prime to m for all n, and this limits the length of the period to at most $\boldsymbol{\lambda}(m)$, the number of integers between 0 and m that are relative prime to m.

Knuth discusses the maximum period of this generator in depth and provides the settings needed in order to achieve this period. In order to reach a **maximum period** of $\boldsymbol{\lambda(m)}$, where $\lambda(m)$ is defined below, the following settings should exist:

1. X_0 is relatively prime to m.
2. a is a primitive element modulo m.

$$\lambda(2) = 1, \qquad \lambda(4) = 2, \qquad \lambda(2^e) = 2^{e-2} \quad if\ e \geq 3.$$
$$\lambda(p^e) = p^{e-1}(p-1), \qquad if\ p > 2$$
$$\lambda\left(p_1^{e_1} \dots p_t^{e_t}\right) = lcm(\lambda(p_1^{e_1}), \dots, \lambda(p_t^{e_t}))$$

Knuth notes that when **$\boldsymbol{m}$ is a prime**, we can reach a period of $m - 1$, which is only 1 less than the maximum period if we were to use an LCG with $c \neq 0$.

2.4.3 Combined MCG (CMCG/CMLCG)

There were various attempts in combining several LCGs in order to construct a new PRNG. The results are PRNGs with larger period and sometimes perform better in some randomness tests, however pose no cryptographic advantage.

We will examine a specific attempt introduced by L'Ecuyer in [20]. The generator is intended to be efficient and portable. His paper discusses the general theory behind the generator and also introduces two new generators, one for 32 bits based machines and one for 16 bits based machines.

The concrete generator suggested for 32 bits based machines is:

MLCG1:
$s_{n+1}^1 = (s_n^1 * a_1)\ mod\ m_1 = (s_n^1 * 40014)\ mod\ (2^{31} - 85)$

MLCG2:
$s_{n+1}^2 = (s_n^2 * a_2)\ mod\ m_2 = (s_n^2 * 40692)\ mod\ (2^{31} - 249)$

Combined MLCGs:
$z_{n+1} = (s_{n+1}^1 - s_{n+1}^2)\ mod\ (m_1 - 1) = (s_{n+1}^1 - s_{n+1}^2)\ mod\ (2^{31} - 84)$

This combined generator achieves a **period** of $p = \frac{(2^{31}-85-1)*(2^{31}-249-1)}{2} \cong \mathbf{2^{61}} \cong 2.3 * 10^{18}$. This generator works as long as the machine can represent all integers between $-2^{31} + 85\ and\ 2^{31} - 85$.

L'Ecuyer's paper elaborates regarding how to implement this generator in a portable manner on different machines. In [21] Schneier provides the exact *C* code of implementing this generator.

2.4.4 LFSR (Linear Feedback Shift Register)

Linear Feedback Shift Register (LFSR) is a shift register whose input bit is a linear function of the previous state. Linear Feedback Shift Registers are prominent building blocks in many cryptographic fields, such as stream ciphers. They are often liked due to the fact they are easy to implement in hardware, produce sequences of large period, have good statistics properties and can be analyzed using algebraic techniques.

An LFSR is comprised of three parts: a shift register, a linear feedback function and a clock which times when the shift occurs. The shift register is a sequence of $\boldsymbol{n}$ bits (in this case we refer to this shift register as an $n\ bit$ shift register).

Each time an output bit is needed, the generator is stepped by shifting all the bits 1 position to the right. The new left-most bit (the **new input bit**) is computed as a function of the other bits in the register. The **output bit** is the bit in stage 0 (the lsb). The **feedback function** is the XOR function (the only linear function of single bits) of certain bits in the register; these bits are called the **tap sequence.**

The **tap sequence** is represented by a polynomial of the form $G(x) = 1 + c_1x + c_2x^2 + \cdots + c_nx^n$ where $c_i \epsilon \{0, 1\}, 1 \leq i \leq n$. If $c_i = 1$ then the i^{th} stage is in the tap sequence. This polynomial is also referred to as the ***connection polynomial*.**

Figure 2 shows an example of an LFSR whose connection polynomial is $1 + x + x^4$.

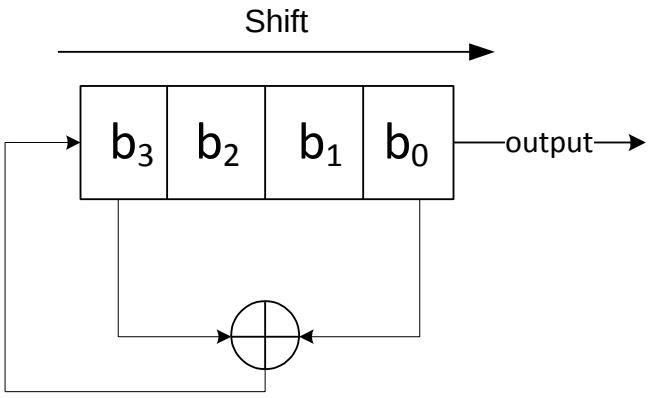


Figure 2 LFSR Example

Maximal Period: the LFSR can result in very long periods; the maximal period of an LFSR is $\mathbf{2^n - 1}$ and it is reached only when its connection polynomial $G(x)$ is a *primitive polynomial* over Z_2 and the initial state is not all zero. These LFSRs are called ***maximum-length*** LFSRs or maximal-period LFSRs. The output of a maximum-length LFSR with non-zero initial state is called an *m-sequence*.

Algebraic form: considering the state of the generator also as a polynomial, much like was shown in the connection polynomial, however here each state bit, b_i, represents an applicable coefficient in the polynomial. Let $S(x)$ be this polynomial; then stepping the generator is equivalent to multiplying by x modulo *G(x)*, i.e., calculating $xS(x) mod\ G(x)$.

Schneier [21] surveys numerous PRNGs (and stream ciphers) that use different LFSR and LFSR combinations.

2.4.5 Lagged Fibonacci Pseudo Random Generators (LFG)

Subtractive random number generator algorithm: this is the suggestion by Knuth [1] (pp. 171-173) for a portable, efficient generator. The emphasis was on a portable generator that only uses integer arithmetic between 10^{-9} and 10^9. The generator is based on the following subtractive recursion:

$$X_n = (X_{n-55} - X_{n-24})\ mod\ m, \quad n \geq 0$$

Knuth argues that the exact value of m is irrelevant; however it does have to be with big magnitude and even. The value suggested by Knuth is $m = 10^9$.

There are several implementations for this generator. Knuth offers an implementation in FORTRAN in his book. There is also an implementation in C described in [2], named *ran3*.

A generalization of the generator introduced by Knuth is a **Lagged Fibonacci Generator** since it is a generalization of the Fibonacci sequence. These generators use an initial sequence $x_1, x_2, \dots, x_j$ and two "lags", j and k.

The recursion of this generator is:

$$X_n = (X_{n-j}\ \$\ X_{n-k})\ mod\ m, \quad 0 < k < j, \quad n \geq j$$

The elements are computer words and '**$**' (the binary operation) is a general binary operation, which can be subtraction, addition, multiplication or the bitwise arithmetic XOR. For addition or subtraction, the x's are either integers $mod\ 2^k$ or reals mod 1. For multiplication, odd gers $mod\ 2^k$.

Period: in [22] Marsaglia shows the **maximal period** of this generator depending on the binary operation shown, while assuming that the modulus used is a power of 2. In [23] we see a good summary of this maximal period, assuming that $j\ and\ k$ are chosen properly and m is defined as

$m = 2^w$ (w being the machine word size). We follow with the summary regarding this maximal period:

$ (Operation)	Maximal Period
$\oplus$	$2^j - 1$
$\pm \, mod \, m$	$2^{w-1}(2^j - 1)$
$* \, mod \, m$	$2^{w-3}(2^j - 1)$

Table 1 Maximal Period

2.4.6 Generalized Feedback Shift Register (GFSR)

Introduced by Lewis and Payne in 1973 [24]; the general idea is to use the ability of the CPU to apply the XOR operation on words. This can be used to run *w* LFSRs in parallel, where *w* is the size of the machine word. Another point of view of this generator is to consider it as an LFG with the operation, *$*, taken as the bitwise XOR operation.

Each LFSR can be considered as one of *w* independent channels. The GFSR recurrence follows:

$$X_n = (X_{n-p+q} \oplus X_{n-p}), \qquad n = p, p+1, \ldots$$

Where $X_n \in \{0,1\}^w, n \in N$, $x^p + x^q + 1$ is a primitive polynomial and *p* and *q* are constants, *p* > *q*.

One should take care with the initialization of the generator. In [24] Lewis and Payne suggest an initialization method which gives the sequence some desirable statistical properties.

The main merits of this generator are:

1. The generator is fast. Generation involves few machine operations per generator step.
2. The generator can achieve an exceptional long period, without dependence of the machine word size. If *p* and *q* are chosen properly the generator achieves a period of $2^p - 1$.
3. The implementation is portable, i.e., is independent of the machine word size.

In [22] Marsaglia wonders why this generator has been given such serious consideration due to the fact that generators with addition or subtraction as the chosen binary operation have better statistical properties and longer periods. In [25] we have a good overview of the drawbacks of this generator:

1. The initialization process of choosing the initial values is critical. Good initialization is rather costly.
2. The generated sequence per channel is known to have poor randomness properties.
3. Although the generator can achieve a long period of $2^p - 1$, it is shorter than the theoretical upper bound period of 2^{pw} (i.e., the number of states possible). In order to achieve a desired cycle length of $2^p - 1$, the generator requires a memory of *p* words.

2.4.7 Twisted Generalized Feedback Shift Register (TGFSR)

Twisted Generalized Feedback Shift Register (TGFSR) [25,26] addresses all the drawbacks of GFSR: it achieves a period of $2^{pw} - 1$ and removes the dependence of a carefully initialized sequence. Furthermore, it doesn't necessary need the polynomial to be a trinomial.

The recurrence of a TGFSR follows:

$$X_n = (X_{n-p+q} \oplus X_{n-p}A), \qquad n = p, p+1, \ldots$$

Where $X_n \in \{0,1\}^w, n \in N$, $x^p + x^q + 1$ is a primitive polynomial, $A_{w \times w}$ is the *twisting matrix*, *p* and *q* are constants, *p* > *q*. We regard X_{n-p} as a row vector and matrix multiplication is done modulo 2. The multiplication by *A* is called a "**twist**".

This generator solves the above problems of GFSR:

1. Neither special initialization process nor precautions are needed. This is due to the fact that unlike the GFSR, this system is not composed of many independent systems (i.e., many LFSRs) but of one unit is which all bits affect each other.
2. The usage of the "twist" with a carefully chosen *A* improves the randomness property of this system.
3. With a proper choice of *A* the system achieves the maximal period, i.e., $2^{pw} - 1$ (achieves all possible states except the zero state). This means that a desired period can be achieved with the minimal needed size of internal state.
4. This generator has the property of *p-equidistribution*, which means that any non-zero sequence of *p* words appears with the same frequency as the output sequence.

2.4.8 Mersenne Twister

Mersenne Twister [27] is an improved variant of the original TGFSR. It achieves a very long period of $2^{19937} - 1$ and extremely good statistical properties while still being very efficient. Its name derives from the fact that its period length is a *Mersenne prime*.

2.4.9 Blum Blum Shub (BBS)

Most of the PRNGs we've covered until now were mostly intended to be used in simulations and other statistical purposes. In order to provide good PRNGs for cryptographic purposes, special PRNGs were constructed. Despite the fact that we didn't encounter any programming language that has an implementation for cryptographic PRNGs that are not Operating System based we'll describe one of the most popular ones here.

Blum Blum Shub (BBS) [17] is a generator whose security properties are based on the computational difficulty of integer factorization. The biggest caveat of this generator is that it is very slow. Following this, it is not appropriate for high performance environments and simulations.

The recurrence of BBS follows:

$$X_{n+1} = {X_n}^2 \bmod M$$

Where $M = pq$ is the product of two large **prime** numbers that are congruent to $3 \bmod 4$. The pseudo-random sequence generated by this generator is the sequence of bits $b_0 b_1 \ldots$ obtained by setting $X_{i+1} = {X_i}^2 \bmod M$ and extracting the bit $b_i = parity(X_i)$. The seed X_0 should be an Integer that is not 1 or divisible by M. Usually the parity is taken to be the least significant bit.

The generator is secure as long as the factoring problem remains hard.

2.4.10 PRNGs in Standards

Recently, there have been some attempts to standardize the implementation of pseudo random number generators. The most comprehensive standard is NIST's Special Publication 800-90 [28] that exclusively addresses the need of generating pseudo random numbers. Another notable standard that describes concrete PRNG implementations exists in Appendix 3 of the FIPS-186 DSS (Digital Signature Standard) [29]. We will briefly describe the key components of each standard.

2.4.10.1 NIST 800-90

This publication is a relatively new publication. This standard is the basis of the new Windows Random Number Generator that is implemented in Windows versions higher than Windows Vista SP1. This Standard has complete details of what is the design of a deterministic random bit generator (DRBG), how to deal with errors, when to reseed, which seed sources to use, requirement of the ability to "personalize" a random stream etc.

The algorithms presented there are analyzed using similar security properties that were mentioned in 2.2. Furthermore, per each recommended generator they present detailed guidance regarding maximum requests between reseeds, maximum entropy inputs and more.

They present algorithms that are based on hash functions such as SHA-1 and HMAC as the generator function, generators that are based on block ciphers such as AES [30] and generators that are based on number theoretic problems such as dual ecliptic curves. Dan Shumow and Neils Ferguson, in [31], showed a backdoor in the latter.

2.4.10.2 FIPS-186 DSS

Unlike NIST's 800-90 standard this publication doesn't solely address the objective of generating random numbers. This standard describes the DSS (Digital Signature Standard) implementation; however for proper implementation of DSS, random numbers are needed. This standard addresses this need by suggesting a pseudo random number generator based on SHA-1 [32] in Appendix 3 of the document.

The Standard presents two PRNG implementations: one that is based on SHA-1 and another that is based on DES [33]. The former is the basis for the previous version of the Windows Random Number Generator that is described in section 3.8. The algorithm uses a one-way function *G(t, c)*, where t is 160 bits, c is *b bits* ($160 \leq b \leq 512$) and G(t, c) is 160 bits. The algorithm also optionally supports a user provided input. In the original publication, the PRNG was specifically described to be used with the DSA (Digital Signature Algorithm); however in a change dating October 5th 2001, the authors also provided a general purpose variation of the algorithm.

In [34] we see a cryptanalysis of the DSS algorithm in case an LCG PRNG was used.

3 Related Work

There is much published on the topic of pseudo random number generators. In this section we'll describe some important related work of analysis and attacks of popular PRNGs. A very good overview of PRNG bugs and suggestion for a secure construction is written by Peter Gutmann [35]. Schneier and Kelsey [6] have a good enumeration of different attack vectors against PRNGs. The links in [36] comprise a thorough list of references that relate to cryptographic PRNGs.

Two of the most important analyses that we'll show are the analyses of the Linux Random Number Generator and the Windows Random Number Generator. We will refer to these works in this paper whenever we'll show that an implementation uses an OS (Operating System) based generator. Due to the fact that our work mostly targets software developers we will also describe popular attacks on software systems and applications that had an ill-implemented or ill-designed PRNG.

3.1 The RANDU PRNG

One of the most infamous PRNGs ever designed. This generator was available as a scientific subroutine for the IBM Mainframe computer (System/360 computer) since the early 1960s and its use soon became widespread.

This PRNG had extremely bad statistical properties due to the ill parameters chosen to implement it. Kunth has a thorough discussion in [1] (pp. 104) regarding this generator and refers to this generator as *"…really horrible"* mentioning that this generator had actually been used on applicable machines for about a decade and saying that *"…its very name RANDU is enough to bring dismay into the eyes and stomach of many computer scientists!"*

The generator is defined by the following recursion:

$$X_0 \text{ is odd}, \qquad X_{n+1} = (65539X_n) \bmod 2^{31}$$

This is an MCG with badly chosen parameters, thus not achieving the full expected period and has some very distinctly non-random characteristics.

Much was then studied regarding the choice of parameters for an MCG and specifically the parameters in the RANDU generator; most notably is the work of Marsaglia in [37], the work of Knuth in [1] and the comprehensive analysis conducted in [38].

The ramifications of the statistical problems discovered in this generator were tremendous. Some[2] even say that due to the widespread of this generator much research during the 1970s in fields that needed random numbers (e.g., simulations) is less reliable than it might have been.

3.2 Netscape SSL Attack

As mentioned in 2.1, SSL's security relies heavily on random numbers – the secret key, *master secret,* is generated using a PRNG. In 1996 [39] a weakness in this PRNG's seeding process as implemented in the Netscape browser's was discovered.

The seeding process of the PRNG that was used in Netscape's SSL implementation as described in [39] follows:

```
1    (seconds, microseconds) = time of day; /* Time elapsed since 1970 */
2    pid = process ID;  ppid = parent process ID;
3    a = mklcpr(microseconds);
4    b = mklcpr(pid + seconds + (ppid << 12));
5    seed = MD5(a, b);
```

Figure 3 Netscape SSL Seeding Algorithm

[2] http://en.wikipedia.org/wiki/Pseudorandom_number_generator#Problems_with_deterministic_generators

Effectively, the seeding entropy sources are the time of day (seconds and microseconds parts), current process id (pid) and parent process id (ppid). The functions *mklcpr* and *MD5* are shown only for completeness of the code but have no security significance.

The authors attack these entropy sources in the following manner:

1. seconds – easiest to find out; our attacker could use a sniffing tool to guess the seconds component.
2. microseconds – relatively easy to guess using brute force since there are only $10^6 \cong 2^{20}$ microseconds in a single second.
3. pid, ppid – If we assume that our attacker has access to the attacked machine, we can assume she can see these variables (e.g., using the *ps* command on a *NIX machine). If we assume that our attacker doesn't have access to the attacked machine we need to make some more observations: First we observe that due to the shift operation showed in line 4 in the pseudo-code we have only 27 unknown bits, which is not the theoretical value of 30 unknown bits since *p/pid* on *NIX machines is 15 bits long. The authors continue to narrow this down by observing that *ppid* is usually just a bit smaller than *pid* or even has a constant value of 1 (the *init* process); this leads to *pid* and *ppid* having just slightly more than 15 bits of entropy. Last, process information can often leak from applications, thus contributing to the fact that this is not a good secret.

This gets us to an efficient attack of between 20 to 47 bits of the seed's information. Netscape fixed this vulnerability in the new version of its browser.

3.3 Predictable Session Keys in Kerberos V4

Kerberos [40] is a security protocol designed to allow entities communicating over a non-secure channel to provide their identity in a secure fashion.

Kerberos relies on a trusted party called *Key Distribution Center* (KDC) that is responsible for maintaining a database of secret keys per each entity in the network. This secret key is only known to the client and to the KDC and is based on a shared secret (e.g., password) between the two. The KDC then uses session keys to provide the client of tickets to services that she is eligible to use. These tickets are used in the remaining of the protocol when interacted with the Service Server (SS).

As shown in [41] , the (session) key generation algorithm used in version 4 of the protocol was flawed and allows prediction of these session keys. Much like in the Netscape attack (section 3.2) the flaw was in the seeding algorithm of the PRNG. The initialization was based on the *time of day*, *process ID* and *machine ID*, which have very limited entropy resulting in an applicable brute force attack. The pseudo code of the generator used is shown in Figure 4.

```
1    srandom(time.tv_usec ^ time.tv_sec ^ getpid() ^ gethostid() ^
            counter++);
2    key=random();
```

Figure 4 Kerberos V4 Generator

3.4 Attack on Apache Tomcat's Session ID Generation

Gutterman and Malkhi [42] published an analysis and concrete attack of the session ID generation in the Apache Tomcat [43] Java servlet container implementation.

To better understand the implications of their result, some background is needed: HTTP is a stateless protocol – this makes the protocol relatively easy to implement and contributed greatly to its huge popularity. Due to this property, mechanisms that allow stateful browsing were created; two of them are the cookie mechanism and URL rewriting. This stateful browsing is what allows

websites to save our shopping carts when we buy online, save our customer preferences, personalize our browsing experience and more.

A special cookie is the session cookie, in which a web server typically sends a cookie containing a unique session identifier. This unique session identifier has to be randomly chosen to not allow others to impersonate us. The technical term for an attack where an attacker impersonates an eligible user by impersonating her session is called *session hijacking*.

The authors found a weakness in the seed implementation of the PRNG used to generate session IDs in Apache's Tomcat server. The weakness that they found holds even if the implementer decided to use the more secure Java PRNG *java.security.SecureRandom*.

The attack showed that the amount of random bits of the seed used isn't as big as expected. They showed that the seed was constructed from a timing based parameter, the system time in millisecond, and from the *toString()* value of org.apache.catalina.session.ManagerBase.java. They continued to show that the toString()'s unpredictability was reduced to the unpredictability of the returned value of *hashCode()*. On Windows machines the *hashCode()* implementation uses an LCG to generate the hash-code value. They showed that this value doesn't add more than 8 unknown bits to the seed.

Their final result is that the amount of unknown bits we have in the seed ranges from 32 to 43 bits. Another contribution of this work is a novel approach of space-time [44] tradeoff to effectively attack PRNGs, which was demonstrated on the attack above.

Their attack procedure was that with the usage of Space-Time tradeoff the attacker guesses all the bits of the seed and matches the result to real session-ids. After matching, the attacker recovers the state of the PRNG and from now on she is synched with the generator's state allowing her to easily hijack sessions.

3.5 Identical NFS File Handles

An infamous example of a **programming bug** in the seeding process of the PRNG used to generate NFS (Network File System) file handles in the Sun OS NFS implementation, the attack is described in [45].

We will start and briefly explain what NFS handles are and why they need to have random properties. NFS is a protocol originally developed by Sun in 1984 that allows a user on a client machine to access files that reside on a different machine (server) over the network in a similar way to accessing files on the local storage/machine.

When a client wishes to access a remote file on the server machine it sends a request with the desired NFS file handle to the server. This NFS file handle is the identification of the object that the client wishes to access. The client receives this handle in the first time he wishes to access this object, by using the *mount* request. The server checks that the client has permissions to access this file system and returns the handle, if permission is granted.

Every client that has a valid file handle can interact with the NFS server, while checks are mostly done during the *mount* operation. In order to make sure that clients go through the *mount* operation, where permissions are checked, the file handles are random to prevent malicious clients guessing a valid file handle.

Whenever a file system is created a program called *fsirand* is run to initialize the file handles with pseudo random values. This implementation initialized the generator with the process ID of the initializing process and the time of day. Due to an implementation bug, the time of day value was never initialized. This meant that the time of day variable contained predictable garbage values, depending on the system architecture used. Effectively only the process ID was used for the initialization of the generator. This process ID was also highly predictable, as most deployments

used the *suninstall* installation procedure to install the initialization program, thus having the same (predictable) process ID.

The result is that many systems ended up using identical NFS file handles. Venema, in [45], described this as "Every other house in the street did have the same keys to the front door".

3.6 Online Poker Exploit

Another good example of a programming error in the usage of a PRNG that led to crumbling of an entire application is seen in [46]. The authors demonstrated an exploit of the shuffling algorithm used in an online poker application by *The Planet Poker Internet cardroom (http://www.planetpoker.com)*.

The authors analyzed the algorithm used in the application to shuffle the deck of cards before each round of Texas Hold'em Poker. The algorithm was published by the software company who developed the algorithm as to show that indeed their algorithm is fair. Their published algorithm follows in Figure 5 Flawed Deck Shuffling Algorithm. The code was implemented in the Pascal programming language [47].

```
1    // Initialize the deck of cards
2    for ctr in 1..52 do
3     cards[ctr] = ctr;
4    end
5
6    randomize;    // initialize the PRNG based on system clock
7
8    // Randomly rearrange each card
9    for ctr in 1..52 do
10    rnd = random(51)+1;
11    tmp = cards[rnd];
12    cards[rnd] = cards[ctr];
13    cards[ctr] = tmp;
14   end
```

Figure 5 Flawed Deck Shuffling Algorithm

The authors showed bugs in the above algorithm that harm the equal distribution of the shuffled deck of cards. Bugs such as the fact that the current card would never be allowed to be swapped with the last card in the deck (ctr=52). This is due to the fact that *rnd* in line 10 would never get to the value of 52. This is because that Pascal's *random(n)* returns a value between 0...(n-1). While this indeed harms the equal distribution of the shuffled deck, the exploit that allowed them to entirely break the application came from the PRNG usage. We follow with an explanation of this exploit.

First they observed that the number of permutations in a deck of cards is equal to $52! \cong 2^{226}$. However the state of the PRNG used is 32 bits long, thus it has only 2^{32} states. They reduced the search space even more by observing that *randomize* initializes the seed with the amount of milliseconds that passed **since midnight**. This reduced the search space even further to $86{,}400{,}000 \cong 2^{26}$ (number of milliseconds in a day). The final nail in the coffin was their final exploit.

By synchronizing the clock of their attack machine and the Poker server they were able to reduce the attack space to a mere $200{,}000$. This is a trivial search space and after a couple of seconds on a regular PC and after seeing just 5 cards in the round they were able to synchronize with the PRNG. From here on now, the application was broken as they knew exactly which card every player got.

The authors further gave recommendations on how to fix the algorithm by using a simpler algorithm, switching to a PRNG with a larger state and not using the system clock as the seed.

3.7 Linux Random Number Generator (LRNG) Analysis

Linux has gained popularity in the last few years in various verticals; most notably in the Super Computer vertical it has a whooping share of 91% [48]. In 2006, Reinman, Gutterman and Pinkas [49] published a comprehensive analysis and a new attack on the forward security of the PRNG used in the Linux kernel (LRNG). We will briefly summarize the structure and algorithm of this generator as described in their paper and their attack results.

The LRNG is an entropy based PRNG [50] which is comprised of three asynchronous stages: (1) Various operating system entropy is gathered from operating system accessible sources, (2) the entropy is added into a TGFSR-like pool using a mixing function, (3) the random bits are extracted from the applicable random pools.

The internal state of the generator is kept in three pools: *primary, secondary* and *urandom*, whose sizes are 512, 128 and 128 bytes, respectively. Entropy collected in stage (1) is added **only** to the primary pool and random bits are extracted from either the secondary or the urandom pool. When necessary, entropy is added from the primary pool to the secondary and urandom pools.

In order to estimate when entropy is needed to be added, either to the primary pool or the other two, the implementation holds an entropy estimation counter per each pool. These counters hold how many bits that are considered random currently exist in the applicable pool.

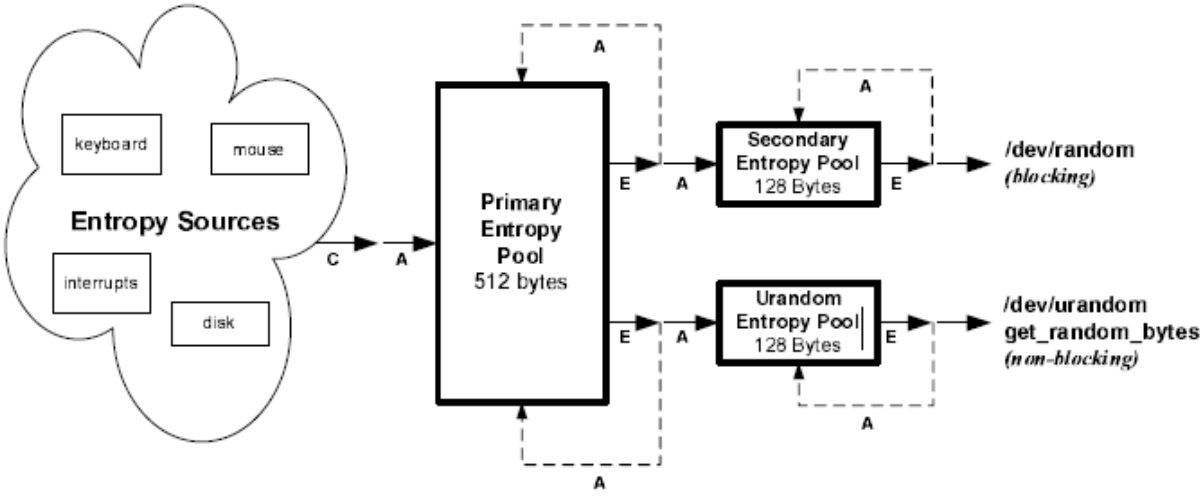


Figure 6 LRNG Structure (taken from authors' paper)

Figure 6 (taken from the LRNG paper [49] page 5) describes the flow and structure of the LRNG. The *C* stands for collection of entropy from the various entropy sources. The *A* stands for addition of entropy to each of the pools. *E* stands for extraction of either entropy from the primary pool or extracting random bits using one of the applicable devices. The dotted line represents a feedback that occurs whenever entropy is extracted from a pool.

There are two devices for extracting random bits from the generator. One is */dev/random*, which is a blocking interface. If one tries to extract random bits from this device, while the entropy estimation for its pool (the secondary pool) is 0, the device would block until sufficient entropy was added from the primary pool. Achieving a property where the random bits extracted from this device should always be "good", while potentially having performance problem for the user due to the blocking nature of the device. The second device is the */dev/urandom* device, which isn't a blocking device. This device would return from its pool (the urandom pool) as many random bits as the user wanted. The third interface that is used by the kernel code to extract random bits is the

get_random_bits function which also returns random bits from the urandom pool but not by reading from the */dev/urandom* device.

The authors show an attack on the forward security of the LRNG with an overhead of 2^{64} in most cases and an overhead of 2^{96} for all other cases. This is a big improvement over the brute force attack that would need an overhead of 2^{1024} for the case of a 128 bytes pool.

Another contribution of the paper is noting a potential DoS attack in case a malicious user would read excessive random bits from the */dev/random* interface. Furthermore, the paper shows the problem of having this generator used in settings that have very limited entropy sources, such as the Open-WRT [51] platform.

3.8 Windows Random Number Generator (WRNG) Analysis

According to [52] the Windows Operating System has a 90% market share; this brings the PRNG implemented in the Windows operating system to be the most used PRNG of all operating systems. Unlike the LRNG (section 3.7), whose source code is readily available, the Windows PRNG (WRNG) is behind closed source.

In 2007 Leo, Gutterman and Pinkas [53] published a detailed analysis of the WRNG of Windows 2000 and Windows XP (with SP < 3) machines. They further showed attacks on the forward security of the generator, which was fixed in Windows XP SP3 [54]. We will briefly summarize the structure and algorithm of this generator and their attack results.

The WRNG is an entropy based PRNG, which uses the algorithm specified in **FIPS-186-2 appendix 3.1** [29] construction with the use of SHA-1 as the G function. There is also use of RC4 when getting entropy from the system. The pseudo code, as taken from the authors' paper, for the main loop of the WRNG follows.

```
1    // output Len bytes to Buffer
2    while (Len > 0) {
3      R = R XOR get_next_20_rc4_bytes();
4      State = State XOR R;
5      T = SHA-1'( State );
6      Buffer = Buffer.concat(T);     // concat denotes concatenation
7      R[0..4] = T[0..4];             // copy 5 least significant bytes
8      State = State + R + 1;
9      Len = Len – 20;
10   }
```

Figure 7 WRNG Main Loop – CryptGenRandom(Buffer, Len)

The **output** of the generator, as can be seen in lines 5-6, is **20 bytes** in size, after the invocation of the SHA1 function. We continue to loop until the buffer is filled with *Len* random outputs from the SHA1 function output.

The function **get_next_20_rc4_bytes** is the function invoked in order to get 20 bytes of random bytes that are fed to the generator. This function is implemented using 8 instances of an RC4 stream cipher operated in a round-robin manner. The ciphers are **initialized** (rekeyed) using system entropy in a **synchronous** way in two situations: (a) at the beginning of the algorithm, (b) if we received 16K Bytes of data from this cipher instance.

The function's algorithm, as described in [53], follows:

```
1    // if | output of RC4 stream | >= 16 Kbytes then refresh the state
2    while (RC4[i]. accumulator >= 16834) {
3      RC4[i]. rekey (); // refresh with system entropy
4      RC4[i].accumulator = 0;
5      i = (i+1) % 8;
6    }
7    result = RC4[i].prng_output(20);
8    RC4[i].accumulator += 20;
```

```
9     i = (i+1) % 8;
```

Figure 8 get_next_20_rc4_bytes()

The implementation uses various system entropy sources to refresh the RC4 instances. The complete list of the entropy sources is available in the authors' paper. Each RC4 is rekeyed with entropy of up to **3584** bytes in a process that also involves utilizing a hash function and additional cycles of RC4 ciphers to produce the actual RC4 key.

The **state of the generator** is dictated by the two variables, *R* and *State* and the states of the 8 *RC4* generators, which each state is 256 bytes long. Both variables, *R* and *State,* are 20 bytes in size, so we can conclude that the state size is $40 + 8 \cdot 256 = 2088\ bytes$.

The authors proposed attacks over the forward security and backward security of this implementation. Both attacks assume the attacker has knowledge of the state at a specific time. By observing the application memory space Leo showed that we can (relatively easily) get the values of: *State, R* and the 8 *RC4* internal states. The paper also discusses why getting the state is relatively easy, mainly due to the fact that, unlike the *LRNG,* this generator is implemented purely in user-mode. We continue to briefly summarize these key findings.

Backward Security: observing that (a) the entropy is refreshed only after the generator produces $8 \cdot 16{,}384$ Bytes, or 128KB of output and (b) the rest of the algorithm is deterministic leads to the fact that **between *RC4* rekeying** the generator has **no backward security** property what so ever.

Forward Security: adding to the above observations the fact that RC4 is **not a one-way function** and has no forward security property, Leo showed that with an overhead of $\mathbf{2^{23}}$ we can break the forward security property of this generator. Moreover, he showed that if we allow ourselves to assume that we can get the values of *R* and *State* at some point in the past, we can get an attack of ***O(1)*** operations. This achievement is due to the fact we only need to invert *RC4*.

Attacks between rekeying: the paper notes that the attacks on the Forward and Backward security of the generator are only applicable between *RC4* re-keying, since after this step the RC4 states are re-initialized. At first this assumption seems harsh, however considering that re-keying occurs every 128KB of output it is actually a serious flaw in the generator. Mostly considering the fact that, according to the paper, this amount of random data is equivalent to 600-1200 *SSL* connections. Considering an average Windows user, this is certainly many connections that need to be performed before the re-keying process takes place.

4 Analysis Methods

In this section we introduce the general structure of our analysis per each programming language in question. We will also use this section to outline common assumptions and common techniques that will be used throughout the analysis.

4.1 Notations/Jargon

The following are notations and general jargon that we will be using throughout the analysis:

- **Generator, PRNG** – we will be using these terms interchangeably to refer to the analyzed generator.
- **Variant, flavor** – all programming languages have more than one PRNG implementation. We will refer to these alternatives as variants or flavors. In some languages even the same generator can have multiple settings that affect its security. We also refer to each setting as a different generator flavor, when applicable.
- **Code segments** – code and pseudo-code segments are designed as following:

```
1    printf("Hello World");
```

- **Period, cycle length** – we will use these two terms interchangeably to refer to the maximal period that an analyzed generator has.
- When discussing bits in a bit array $b_n, b_{n-1},\ldots, b_0$ then the **0 indexed bit stands for the lsb bit**.

4.2 Assumptions

The following are assumptions that we make in all of our programming languages:

- **Architecture** – we assume our architecture is based on 32 bit architecture.
- **Operating Systems** – our analysis covers the two most popular operating systems Microsoft Windows [55] and Linux [56]. If there are differences between versions due to different implementation we will note when applicable. In some languages, e.g., PHP, we decided to present a complete analysis only for the Linux platform; this due to the fact that most PHP deployments happen on Linux platforms.

4.3 Common Analysis Structure

We will provide our analysis per each programming language using the following loose structure:

- **Introduction** – each programming language will have an introduction section. In this introduction section we provide background information for the analysis. Information such as the popularity of the programming language, the different PRNG implementations that exist in this language, version information that relates to our analysis, scope of our analysis and applicable resources. Here we will also state the specifics of how the analysis was conducted in terms of source code accessibility.

Per each flavor of PRNG that exists in the programming language we provide the following:

- **Introduction** – miscellaneous and introductory information that relates to this specific flavor.
- **Design Space** – here we discuss aspects that relate to software design of this implementation. We specify the source files, header files, class files, functions and overall design that the implementation utilizes. Here we also explain the API that a developer can use to interact with this generator.
- **Under the Hood** – concrete implementation details, including the PRNG properties of the generator. Here we will provide detailed explanations of the algorithm used, including pseudo code in most programming languages. Each programming language would **minimally** contain the fol-

lowing information in this section: What is the theoretical PRNG behind this implementation? Is there a way for the user to set the seed? What is the default seed implementation (if there is any)? What is the size of the state? What is the size of the seed? What is the period of this generator? Is this generator entropy based?

- **Security Properties Analysis** – this section holds a detailed analysis of the security of this generator. We follow the security requirements explained in 2.2 that are: pseudo-randomness, forward security and backward security. Here we will describe the various attacks that we found in the applicable implementation. Where applicable we will also cover the security of the default seed implementation and address the security of the seeding operation as a whole.

4.4 Attack Vectors and Attack Assumptions

Analyzing a programming language API without using a specific application in mind or an application as an attack target can be hard. If we were to give our attacker too much strength then most generators would be easily broken, e.g., since most generator run in user space an attacker that has access to the machine can almost always access the concrete state, which would have made our attacks trivial. The following are the attack vectors and attacker strength assumptions we used:

- **Cipher text attacks** – we only assume our attacker has access to outputs (or sometimes part of the output) of the generator. We do not allow our attacker to have access to the machine, nor the ability to change the state or parameters of the generator (although where applicable we will state weakness if those parameters are easily changed by an attacker with access to the machine).
- **Space-time tradeoff attacks** – space-time tradeoff is a technique that allows an attacker to balance between the space and the time of her attack. Gutterman and Malkhi provide a general scheme for the use of space-time tradeoff in PRNGs in [42]. We will not explicitly mention how to use this technique in our attacks even in places it can be utilized.
- **Consecutive outputs** – some attacks require getting consecutive outputs from the generator. In most attacks the consecutive outputs assumption can be replaced with an assumption that we can know the applicable position in the random stream. We note that this assumption isn't very harsh considering that there are many applications where getting consecutive outputs is relatively easy.
- **Time based attacks** – many seed implementations use time (or clock) values in order to seed the generator in their default seed implementation. We will see that this type of seeding has low entropy under realistic assumptions regarding the server up time, or other relevant parameters.
- **Solving linear equations** – in some attacks we'll use the fact that the generator output gives us linear equations over the generator's state. For example, see the attack described in 5.4.4.2.1.

5 C

5.1 Introduction

There are many PRNG implementations for C. We will concentrate on the PRNGs that are available in the standard C language specification, popular compilers and standard runtime implementations. We will discuss the Microsoft C runtime (**MSVCRT**) [64] implementation and the gnu C library (**glibc**) [58] which is popular on *NIX platforms.

Within these runtime implementations, we will discuss the ISO ANSI-C ***rand()*** family that is available on Windows and *NIX platforms and other families: the **BSD** [59] ***random()*** and **SVID** [60] *drand48()* traditional UNIX PRNG families. The latter are only available on ***NIX** platforms as they aren't available with the default Microsoft runtime. On **Windows** platforms, as part of the security enhancement in the CRT, there is a different flavor of *rand()* called ***rand_s()*** [61]. This variant will also be covered here.

We note that there are many other **3rd party libraries** that implemented other PRNGs, such as implementations that follow the algorithms in [2]. These are out of the scope of this analysis. [62] Gives a very good review on the algorithms presented in [2], as far as randomness and cycle length of these generators.

glibc specific scope: There are several implementation variants in *glibc*; one of which is the ***reentrant* functions** that as a convention have their function names end with a *_r* suffix (as defined in the POSIX standard [63]). This analysis only covers the regular, non-thread safe functions. The main difference of these variants is that the state isn't preserved in global variables accessed by the random functions but instead provided by the user during invocation of the function. However the basic PRNG algorithm remains the same.

Importance of C generators: C is still one of the most popular languages used in the software industry; especially in a performance demanding environment, such as embedded devices and real-time applications. C is a major building block in modern programming languages and technologies: the Java JVM is built partly in C, so is Microsoft's CLR, Perl`s engine is written almost completely in C and so on and so forth. Some of these languages still use the generators that are available in C, either as fallbacks in case other variants can't be used, or even as the default generator to use.

Version information: The **glibc** version that was studied was **glibc-2.5** dated 29/9/2006. The Microsoft **CRT** implementation version that was studied was the one supplied with **Visual Studio 2005** [64].

Structure of analysis: the structure we use for this analysis is a bit different than the one we use to cover other languages. Since the dependence on platform in C is stronger than in other chapters, we will analyze the Windows and the *NIX variants as independent generator implementations.

ANSI C Standard PRNG specification: C is a standardized programming language; it was standardized in 1989 and ratified as ANSI X3.159-1989 "Programming Language C." [65,66]. According to the Rationale document [67], the Committee also noted the requirement of having a pseudo random number generator implemented. They further claimed that the function should generate the best random sequence possible in that implementation (meaning the implementation of ANSI C) and therefore mandated no standard algorithm. Nevertheless, they recognized the value of being able to generate the same pseudo-random sequence in different implementations, and so they published as an example in the Standard an algorithm that generates the same pseudo-random sequence in any conforming implementation given the same seed (can be seen in [67], 4.10.2, p 101). The algorithm is a portable one and is based on the LCG algorithm (2.4.1). Section 7.20.2 in the Standard requires the following:

1. *rand()* function – (a) The rand function computes a sequence of pseudo-random integers in the range 0 to RAND_MAX, (b) the value of the RAND_MAX macro shall be **at least** 32767 ($2^{15} - 1$).
2. *srand()* function - The *srand* function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to *rand*. If srand is then called with the same seed value, the sequence of pseudo-random numbers shall be repeated. If *rand* is called before any calls to *srand* have been made, the same sequence shall be generated as when *srand* is first called with a seed value of **1**.

5.2 Microsoft CRT (MSVCRT) Generators

5.2.1 (ANSI-C) C Standard Built-in Generators (rand() family)

http://msdn.microsoft.com/en-us/library/398ax69y(VS.71).aspx

5.2.1.1 Design Space

API: as required by ANSI-C, the core functions in the CRT that relate to generating pseudo random numbers are ***rand()*** and ***srand*** (used to setting the seed) functions.

Adaptation from BASIC: according to the function comment of *srand* the algorithm is adapted from the BASIC random number generator.

The functions are declared in the *stdlib.h* header file and implemented in the *rand.c* source file.

5.2.1.2 Under the Hood

The **state/seed** of the generator is held in a variable named ***_holdrand*** of type *unsigned long*. Each thread has its own state variable, which is saved in the per-thread data structure named *_tiddata*. *_tiddata* is a *struct* which is declared in *mtdll.h* (the include file for DLL/Multi-thread). This struct also holds various thread related information such as the *thread-id*, thread handle and various other data.

The theoretical PRNG behind it is LCG: the LCG's implementation is as seen in the following pseudo-code:

```
1    seed = seed • 214013 + 2531011;
2    output = (seed >> 16) & 0x7fff; //output is truncated to output
                                        maximum of 32767
```

The recurrence formula of this LCG is:

$$X_{n+1}=(a \cdot X_n + c) \bmod m,\ n>=0,$$

$a = (214013)_{10}$
$c = (2531011)_{10}$
$m = 2^{32}$

Implicit modulus parameter: m is chosen as a power of 2, since the implementation is done with 32 bit unsigned arithmetic the addition of two unsigned numbers is performed with a modulo of 2^{32}.

Output truncation: the output of the generator is bits 16-30 of the generator's state.

Period: the generator has the maximal LCG period of $\mathbf{2^{32}}$ (the size of m). This is due to the fact that the parameters chosen satisfy the requirements outlined in 2.4.1:

1. **m** is chosen as a power of 2.
2. **c** and **m** are relative primes as their *GCD* is 1.
3. Since *m* is a power of 2, its prime factors are **2**; following this **(a-1) =** 214012 is divisible by all prime factors of m.

4. Both **m** and **(a-1)** are a multiple of 4.

State: the state size is effectively **31 bits**. This is due to the fact that the MSb bit of the state is never used. We can see that during stepping of the generator the MSb bit of the state only affects the MSb of the state, and because we don't use this bit in our output this bit actually has no contribution to our generator.

Seed: there is an option to set a seed externally by invoking the *srand* function. The function sets the generator's state to be the function's argument. We note that the fact, which follows from the paragraph above, that only 31 bits of the given seed affect the generator, is not documented.

Default seed implementation: the default seed is initialized to be the constant **1**. This can be seen in the initialization process of the *ptd* structure, whenever a new thread is initialized, in the source file *tidtable.c* line 482, in the function *_initptd*.

Entropy use: this implementation **doesn't** add any entropy to the generator.

5.2.1.3 Properties Analysis

5.2.1.3.1 Pseudo-randomness

Assuming we know the implementation is based on *rand(),* i.e., the generator is LCG with known parameters.

Known Cipher-text attack: we can mount a similar attack to the one outlined in details in 6.3.3.1 and we'll give here only a brief sketch - The attack will require us to find out the missing bits that were truncated before the output was generated. Assuming that a common implementation would require all the **15 output bits** from the generator, we'll find the unknown 16 remaining bits of the seed by enumeration and validation.

Number of outputs needed: given an output, after seeing another verification output we will have $1 + 2^{-15} \cdot (2^{16} - 1) \cong 3$ legal guesses for the internal state. If another output will be used to verify the correct state we'll have only $1 + (2^{-15})^2 \cdot (2^{16} - 1) \cong 1 + 2^{-14}$ valid options. So, we can conclude that when using **two more outputs** for validation, we can find the real state.

Assuming we don't know that rand() is used. We note that we can't mount the Boyar [68] attack in order to try and find out the LCG parameters (and conclude that this is in fact an LCG). This is due to the fact that the output is never the entire state. If we get consecutives outputs, an easy distinguisher is simply to mount the attack above, and then verify with another output.

5.2.1.3.2 Backward Security

None (not entropy based).

5.2.1.3.3 Forward Security

None; since it uses LCG, with knowledge of the current state we can simply reverse the LCG and get to the previous states.

5.2.1.3.4 Default Seed Weakness

The CRT implementation actually doesn't even try to provide with an adequate default seed. In case the user won't set the seed herself, the constant seed will be used, which is obviously not even remotely secure.

5.2.2 rand_s()

http://msdn.microsoft.com/en-us/library/sxtz2fa8(VS.80).aspx

5.2.2.1 Design Space

Part of the efforts of making the CRT more secure that is described in [69], a new convention of function names was introduced. This convention was to add a suffix ***_s*** ("secure") for functions that are now more secure (A good example is the new *strcpy_s* function which is a secure counterpart of the *strcpy* CRT function. This new function takes another parameter, which is the size of the buffer, so it can determine whether a buffer overrun will occur).

rand_s is another one of these new functions, which is a secure alternative for the CRT *rand()* function analyzed in 5.2.1. The *rand_s* implementation is in the ***rand_s.c*** source file and declared in *stdlib.h*.

In order to use this variant one should define, prior to the inclusion statement of *rand_s,* the constant ***_CRT_RAND_S***. This implementation is completely separate than the one of *rand()* and *srand()*, thus it doesn't use the seed set by *srand()*, nor does it affect the state of *rand()*.

Applicable Windows Versions: According to documentation in [61] this variant only works on Windows XP and later. It uses the ***RtlGenRandom*** function, which is defined in *NTSecAPI.h* and available in *ADVAPI32.DLL* in order to invoke the *WRNG* (3.8). The implementation of *RtlGenRandom* is exported as *SystemFunction036* in the DLL above. On Windows XP machines and later *CryptGenRandom* invokes *RtlGenRandom*; according to [70] this was done in order for callers that do not want to load the entire *CryptoAPI* to still be able to call the WRNG.

The API of *rand_s* is different than the one in *rand()*:

```
1    errno_t rand_s(   unsigned int* randomValue);
```

The function receives an int pointer, in which the next random integer will be placed. According to the documentation the function produces a random number in the range *0…UINT_MAX* $((0x\text{ffffffff})_{16} \cong (2^{31}-1)_{10})$.

Misleading MSDN Documentation: there is an inaccurate statement in [69]. According to [71] the .NET Framework equivalent of *rand_s* is *System::Random* class. From our analysis of .NET in the 7.2 chapter we know that the generator implemented in *System.Random* doesn't use the *WRNG* generator.

5.2.2.2 Under the Hood

The function implementation just invokes the function *RtlGenRandom* requesting random value of 32 bit (size of unsigned-integer). As of such its analysis is identical to the one described in 3.8.

5.3 *NIX glibc Generators

5.3.1 Introduction

There are many random number generators that are available in the *glibc* library. All generators are declared in the *stdlib.h* header file. The resolving of different algorithms is based on constants defined by the user.

5.3.2 (ANSI-C) C Standard Built-in Generators (rand() family)

http://www.gnu.org/s/libc/manual/html_node/ISO-Random.html#ISO-Random

5.3.2.1 Design Space

API: as required by ANSI-C, the core functions are ***rand()*** and ***srand*** (used to setting the seed).

The functions are declared in *stdlib.h* and implemented in *rand.c* and *random.c* (*srand*). The implementation for *rand()* invokes the BSD variant *random()* function and *srand()* is mapped to the *srandom* function.

Following this, the reader in encouraged to see the analysis of the *BSD random()* functions family in the following section (section 5.4).

5.4 BSD C Generators (random() family)

http://www.gnu.org/s/libc/manual/html_node/BSD-Random.html

5.4.1 Introduction

The BSD style generators are available in the *glibc* library as a mean of compatibility for BSD like systems.

Importance of BSD generators: The same generators are also available in the Mac-OS X Operating System, as it is based on BSD. This makes these variants even more important considering the recent popularity of Apple based products, which all of them are built on top of some version of the Mac-OS X. E.g., the popular iPhone (and now iPad) device are built on top of an OS, which is reportedly derived from Mac-OS X [72].

5.4.2 Design Space

The implementation of the *random* functions family resides in *random.c* source file and defined in *stdlib.h*. The implementations of *random(), srandom* and other family functions invoke the implementation of *random_r()*, *srandom_r()* and other corresponding functions.

Output size: the *random()* method returns a **31 bit** value.

The implementation has two basic modes of operation: (1) An **LCG** implementation. (2) An implementation based on **Additive feedback generator** (2.4.6) with 4 different polynomials. We will use the acronym **AFG** during this analysis to indicate the latter.

The state array can be specified by the user using the ***initstate* function**. This function allows the user to specify her desired size of state array and seed. Consequently this will determine the polynomial used for the AFG, and whether the AFG or LCG is used. There is another API function called ***setstate*** that is used to re-set the state array.

For further details regarding the API the reader is encouraged to read the *libc* documentation in [73].

Apart from these functions the operation API is similar to the one of the *rand* functions family. Namely, there's a ***srandom*** function in order to initialize the seed and ***random*** function that is used for stepping the PRNG.

State Array: Each entry in the state pool information array is an *integer*. The implementation uses several pointers in order to manipulate this state array, see 5.4.3.1 and 5.4.4.1 for details.

Generator types: there are 5 types of generators used. The choice between the generators is based upon the amount of information in the state array, i.e., the **length of the state array** as provided in *initState*. This can be seen in the next summary table:

#	Implementation (trinomial)	|Input State|(Bytes)
G0	LCG (N/A)	8 ≤ |state| < 32
G1	AFG ($x^7 + x^3 + 1$)	32 ≤ |state| < 64
G2	AFG ($x^{15} + x + 1$)	64 ≤ |state| < 128
G3	**AFG ($x^{31} + x^3 + 1$)**	**128 ≤ |state| < 256**
G4	AFG ($x^{63} + x + 1$)	256 ≤ |state| < *

* - a state size that is bigger than 256 is truncated to 256.

The actual code that defines these implementations can be seen in 11.3.1. All implementations are implemented in the *random_r()* function, which is implemented in the *random_r.c* source file. The decision which implementation to use is controlled via the *rand_type* variable.

If the user doesn't initialize a state herself, the **default generator** chosen according to the default initialization used, which leads to the **G3** implementation.

The analysis continues in the following structure: we first analyze the *G0* variant and then continue to analyze the variants *G1-G4* as they all share the same algorithm with different parameters.

5.4.3 G0: LCG

5.4.3.1 Under the Hood

The implementation uses the first element of the state array as the LCG's **state**, meaning an *integer* value of **32 bits**.

The theoretical PRNG behind it is LCG: the recursion formula of this LCG is:

$$X_{n+1}=(a*X_n + c) \bmod m,\ n >= 0$$

$a = (1103515245)_{10}$
$c = (12345)_{10}$
$m = 2^{31}$

The actual code of the implementation can be seen in 11.3.1.

Resemblance to ANSI-C example algorithm: this implementation and parameters are the parameters used in the algorithm example in the ANSI C Standard. The only difference is that this implementation allows the output of up to 31 bits, instead of the 16 bits used in the Standard.

Period: the generator has the **maximal LCG period of 2^{31}** (the size of m). This is due to the fact that the parameters chosen satisfy the requirements outlined in 2.4.1.

State: the state size is **31 bits** long, due to the modulus used.

Seed: there is an option to set a seed externally by invoking the *srandom* function. The function simply sets the state to be the seed supplied. It first makes sure the seed supplied isn't equal to 0. If it is equal to 0, the implementation sets the seed to be equal to **1**, as specified in the ANSI-C Standard.

Default seed implementation: as specified in the ANSI-C Standard, the default seed is equal to **1**.

Entropy use: this implementation **doesn't** add any entropy to the generator.

5.4.3.2 Properties Analysis

5.4.3.2.1 Pseudo-randomness

Unlike other LCG implementations that we've covered in this paper, here the output of the generator is simply the state of the LCG. This leads to the following attacks.

Assuming we know the implementation is based on *random()*, i.e., the generator is LCG with known parameters.

Known Cipher-text attack: we notice that if we can get our hands on a complete output of the state, meaning the application will request an output of 31 bits, then our attack is complete and we have the state in our hands. If we don't get a complete output we can still mount a cipher-text attack similar to the one in MSVCRT (5.2.1.3.1). We will always guess the amount of bits we didn't get as output as the bits we need to guess in order to get to the state, and use another output(s) for validation.

Assuming we don't know that random() is used. If we get enough outputs ($log_2(2^{31}) = 31$), we can mount the attack proposed by Boyar [68] in order to find out if the generator is LCG with the known parameters. However we since we just want to verify that the generator is LCG with given parameters, we can simply do it using two consecutive outputs.

5.4.3.2.2 Backward Security

None (not entropy based).

5.4.3.2.3 Forward Security

None; since it uses LCG, with knowledge of the current state we can simply reverse the LCG and get to the previous states.

5.4.4 G1-G4: AFG

5.4.4.1 Under the Hood

The theoretical PRNG behind is AFG: the algorithm uses the *additive number generator*; in [1] pp. 26-28, Knuth discusses these types of generators in detail. The simplified recursion function of the algorithm is:

$$X_n=(X_{n-deg} + X_{n-sep}) \bmod 2^{32}, \qquad n >= 0$$

Where ***deg*** is the degree of the polynomial used and ***sep*** is the separation between the two lower order of coefficients of the trinomial, meaning the distance between *fptr* and *rptr* as seen in Figure 10 AFG Algorithm . For each of the generators, G1-G4, we get the following:

#	deg	sep
G1 ($x^7 + x^3 + 1$)	7	3
G2 ($x^{15} + x + 11$)	15	1
G3 ($x^{31} + x^3 + 1$)	**31**	**3**
G4 ($x^{63} + x + 1$)	63	1

Figure 9 deg, sep assignment per each flavor

The algorithm implementation can be seen in the following diagram:

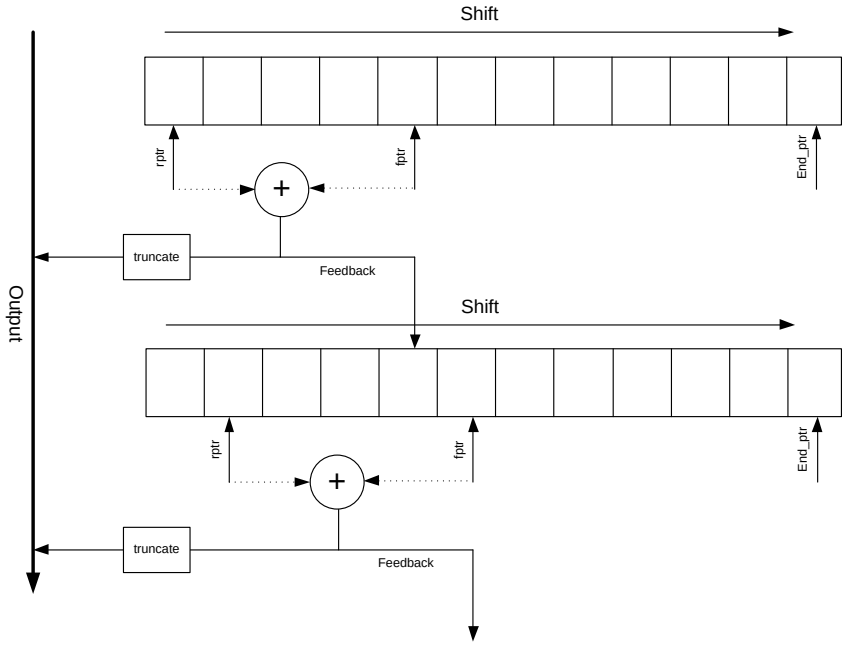


Figure 10 AFG Algorithm Diagram

Implementation details: similar to the implementation in 7.2 (C#), the implementation uses an array of *(signed) integers*. The algorithm keeps three pointers to the array: front pointer, rear

pointer and an end pointer (*fptr, rptr* and *end_ptr* accordingly). *fptr* and *rptr* are positioned in a distance of ***sep*** between them. The actual code of the algorithm can be reviewed in 11.3.1.

Stepping the generator: in each iteration of the generator *fptr* and *rptr* and summed, the summed product is placed where *fptr* points to, to create a feedback. Then the two pointers increment by one, i.e., the register is *shifted*. Incase either of the two pointers reaches the end of the array, by reaching the end pointer, it wraps around to the start of the array.

Output reduction: the function returns the summed product reduced to **31 bits** by chucking the **least significant bit**.

State: the state of the generator is the array of *integers*, which depends on which generator is used. The **state size (|state|)** of the generator is $\boldsymbol{deg * 32\ bits}$; which means 7*32=224 bits, 15*32=480 bits, 31*32=992 bits, and 63*32=2016 bits for generators G1, G2, G3, and G4 respectively.

Inaccuracy of code documentation regarding period length: we note that the code documentation states that the algorithm reaches the period length of $deg * (2^{sep} - 1)$. Furthermore, it states that surely the period for G1 isn't small as $7 * (2^7 - 1) = 889$. According to the literature we found the actual period, when a modulus of the power of 2 is used and the generating polynomial is primitive, is as follows.

Period: since all polynomials used are primitive we know that the least significant bit achieves the maximal period, which is $2^{|state|} - 1$. This means that our generator achieves **at least** this cycle length. As noted by Knuth in p 27 in [1] the period of the entire algorithm is bigger than this since the summation also affects the high order bits (since there is a carry for the summation). According to [23] the cycle is $\boldsymbol{2^{M-1} * (2^{deg} - 1)}$, where *M* is the power of 2 used in the modulus. I.e., we can conclude that the period for G3, G4 is $\boldsymbol{2^{31} * (2^{31} - 1)}, 2^{31} * (2^{63} - 1)$ respectively.

Entropy use: this implementation **doesn't** add any entropy to the generator.

Seed: there is an option to set a seed externally by invoking either *srandom* or *initstate. initstate* initializes the data structures and invokes *srandom* to perform the actual initialization logic. We note that a user can also pass a complete state array using *setstate* function. The seed is a single *unsigned integer*. Here, as in G0, a seed value of 0 is not allowed and 1 is used instead.

Default seed: the default seed is equal to **1.** The initial state array is populated from this value as following.

Initial state generation procedure from the given seed: in order to populate the state array the implementation uses a two-step procedure: (1) invokes an LCG generator on the given seed and fills the entire state array, (2) cycles the entire state array **10 times** by invoking the *random* function. This can be seen in the following pseudo-code:

```
1    state[0] = seed;
2    for i = 1..deg do
3      // X_{n+1} = 16807 X_n mod 2^31,   n > 0
4      state[i] = (16807 * state[i-1]) % 2147483647;
5    end
6
7    for i = 0..deg*10 do
8      random();
9    end
```

Figure 11 State Initialization Code (srandom function)

We note that the actual code also makes sure that result of the LCG won't overflow 31 bits. The actual source code can be viewed in 11.3.1.

5.4.4.2 Properties Analysis

5.4.4.2.1 Pseudo-randomness

As mentioned above, for the sake of the simplicity in the analysis of pseudo-randomness we will consider that the generator used is **G3**, because it is the default generator that most users would use. The analysis is easily extended to other generators by using the different generators` parameters.

Assuming we know the implementation is based on *random* G1-G4: this will give us the information of the algorithm used for the generator and its parameters.

Brute Force: a brute force attack of the state would require searching a space of $\mathbf{2^{992}}$, which is not a feasible search space. By allowing ourselves to get outputs from the generator, we get the following improved attack.

Known cipher-text attack: note that even after getting **31 consecutive outputs** from the generator, we still have $\mathbf{2^{31}}$ **options** for the unknown LSbs of the 31 words of the state that were truncated from the output. To find these bits, we will follow Klein's [74] attack to break the state.

Reminding ourselves that our series is:

$$X_n = (X_{n-31} + X_{n-3})\ mod\ 2^{32}$$

An equivalent representation is:

$$X_{n+31} = (X_n + X_{n+28})\ mod\ 2^{32}$$

Writing out internal state at step **n** as:

$$X_n = 4 * K_n + 2 * L_n + B_n$$

Where K_n represents bits 2…31, L_n is bit 1 and B_n is the LSb bit (bit 0), which is what we are trying to find.

Extracting $\boldsymbol{B_n}$: Observing how X_n advances: if $\boldsymbol{B_n}$ **and** $\boldsymbol{B_{n+28}}$ **are both 1**, then we have a carry from this LSbs to L_n. This means that when we see from our outputs that $L_{n+31} = (L_n + L_{n+28} + 1)\ mod\ 2$, we can surely conclude that

$$(*)\ B_n = 1\ ;\ B_{n+28} = 1.$$

Note that both equations are linear equations in the state bits, for example if we know in this way that $B_{33} = 1, B_5 = 1 \rightarrow\ B_{30} \oplus B_2 = 1, B_5 = 1$.

If $\boldsymbol{B_n = 0}$ **or** $\boldsymbol{B_{n+28} = 0}$, we don't have a carry, thus if $L_{n+31} = (L_n + L_{n+28})\ mod\ 2$, we can only know that

$$(**)\ (B_n = 0 \textbf{ or } B_{n+28} = 0).$$

Note that, like in the previous case, we are actually getting constraints on the state bits, for example if we get that $B_6 = 0\ or\ B_{34} = 0 \rightarrow\ B_6 = 0\ or\ (B_3 \oplus B_{31}) = 0$.

Klein follows and shows that we would need a minimum of extra 38.27 outputs. This is due to the fact that beyond the initial 31 outputs the distribution of this **carry bit** is ¼:¾, thus by applying the binary entropy function we get:

$$H_2\left(\frac{1}{4}\right) = -\frac{1}{4}log_2\left(\frac{1}{4}\right) - \left(1 - \frac{1}{4}\right)log_2\left(1 - \frac{1}{4}\right) \cong 0.81\ bits\ of\ infomration$$

Meaning we would need 31+31/0.81≅69.27 outputs. However this only holds if the variables were independent; this is not the case here, so we would need some more outputs (equations) in order to have all the information to find B_n. Klein argues that we would need 80-100 outputs in order to get to a single solution. So we can conclude that if we were to get **80-100 consecutive outputs**, we would be able to **reconstruct the entire state**.

After showing in the discussion above that 80-100 outputs should be enough to get the state (information wise), we'll now show how we can get to the state from those outputs. We have two alternatives:

1. Use a brute force approach over the 2^{31} space (of the missing B_n bits) and for each option verify that the generator indeed produces the 80-100 real outputs. We know from the information theory reasoning above that only one such option will pass this validation.
2. A more efficient way is to first solve the linear equations in (*) above, and get all the candidates for $B_0 \dots B_{31}$ that fulfill the (*) constraints. Following this, we will eliminate false solutions using the constraints in (**) above.

Assuming we don't know the implementation is based on *random*. We could just try and mount the attack above in order to distinguish this generator`s output from a random output.

5.4.4.2.2 Backward Security

None (not entropy based).

5.4.4.2.3 Forward Security

None – if we have the state in our hands, we can reconstruct the subtraction equations in order to get to a previous state.

5.4.4.3 Seed Weakness

State initialization weakness: we note that the procedure in the initialization process is completely reversible. If we managed to extract the state information using the attack outlined above (section 5.4.4.2.1) we are able to get to the initial seed by reversing the initialization process. This, by itself, can lead to even greater exposer, in case the seed is supposed to be secret (as seeds tend to be). Klein mentioned this in his attack that by reversing to the seed from the revealed state, the attacker can have a coarse indication as to the amount of DNS outgoing queries sent.

Brute force: the seed is only a 32 bit integer (before the warm up phase that expends it to 31 words), so if we get an output and we know the number of output iterations since initialization we can mount a brute force attack that will use this output for validation. Note that here we need to know how many times the generator was stepped, as opposed to the attack above that could use any 100 consecutive outputs.

Default seed weakness: the default seed is constant, thus has no entropy what so ever.

5.5 SVID C Generators (rand48() family)

http://www.gnu.org/s/libc/manual/html_node/SVID-Random.html

5.5.1 Introduction

This family of functions is intended for compatibility with the SVID standard [75]. As it name suggests these functions use 48 bits of state size.

5.5.2 Design Space

There are several flavors of functions for the caller to choose from. The functions differ mainly by the way the random bits are returned, e.g., double, long etc.

Two distinct types of functions exist: one that generates output from a global state of the generator and another that allows the user to explicitly pass the entire state of the generator. All these variants use the same generator algorithm.

API: the various API functions for **generating random** values are:

1. **drand48** – Returns a non-negative, double floating point value in $[0.0, 1.0)$.
   erand48 – Same as the above, only allows the user to specify the complete state.
2. **lrand48** – Returns a non-negative, long integer in $[0, 2^{31})$.
   nrand48 – Same as the above, only allows the user to specify the state.
3. **mrand48** – Returns a signed, long integer in $[-2^{31}, 2^{31})$.
   jrand48 – Same as the above, only allows the user to specify the state.

For further details the reader is encouraged to refer to the documentation in [60].

Source files: All of the above functions reside in separate implementation files with the filename as the name of the function, e.g., drand48 resides in *drand48.c*. The actual implementation of the generator resides in *drand48-iter.c* source file and this was used as the source for our analysis. The main function is named accordingly *__drand48_iterate* – this is an inner function which is not exported to the user.

Like in the BSD variants (5.4) there are also *reentrant* function variants that end with the *_r* suffix. Their implementation is not covered explicitly in this analysis as they share the same generator algorithm.

Initialization: there are several functions that can be used in order to set/initialize the generator. The functions differ from one another by the amount of control the user has in initializing the generator and consequently the amount of information the user has to supply for the initialization process. Below is a quick summary of the various initialization functions.

1. ***srand48*** *(long int seedval)* – seeds the generator. Receives a 32 bits seed value.
2. ***seed48*** *(unsigned short int seed[3])* – seeds the generator allowing setting the entire 48 bits of the state.
3. **lcong48** (unsigned short int param[7]) – allows complete control on the generator's state and parameters. We note that this level of control, although good if one wants to an entirely different configuration for the algorithm can cause abuse, since the algorithm parameters need to follow strict requirements to guarantee adequate randomness and a full period.

Initializers source files: All of the above functions reside in separate implementation files (C files) with the filename as the name of the function.

5.5.3 Under the Hood

State structure: the implementation uses three *shorts* (2 Bytes) in order to represent the generator's state. The structure used in the various functions is *drand48_data*, which is defined in *stdlib.h.*

This structure holds the current state, previous state and various parameters of the generator. The structure`s source code can be seen in 11.3.2.

The theoretical PRNG behind it is LCG: the LCG's implementation's source code is:

```
1    X = (uint64_t) xsubi[2] << 32 | (uint32_t) xsubi[1] << 16 | xsubi[0];

2    result = X * buffer->__a + buffer->__c;

3    xsubi[0] = result & 0xffff;
4    xsubi[1] = (result >> 16) & 0xffff;
5    xsubi[2] = (result >> 32) & 0xffff;
```

Figure 12 Rand48 Algorithm Code

xsubi[] is the array that holds the state information for the generator. The LCG formula is performed as usual with *X* as the variable that holds the state. The translation of the array state to X is:

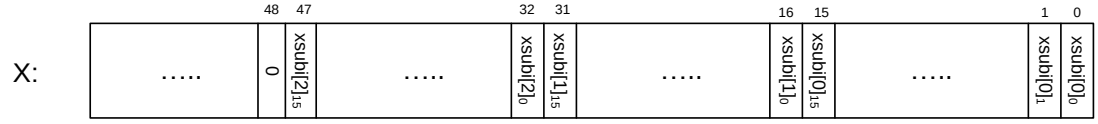


Figure 13 Diagram of Translation from xsubi Array to the State Variable X

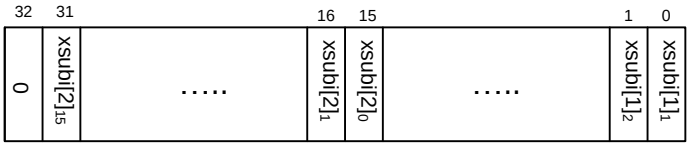
The recursion formula of this LCG is:

$$X_{n+1} = (aX_n + c)\ mod\ m, \quad n \geq 0$$

$a = (25214903917)_{10}$
$c = (11)_{10}$
$m = 2^{48}$

Parameters choice: this choice of (a, c, m) guarantee a LCG with a maximal period.

Output truncation: the output of the LCG implementation depends on which function from the API we chose to use. In order to simplify the analysis, we will consider only the function of *lrand48/nrand48* that returns **31 bits** which represent an unsigned integer. The output that is returned is the 31 MSB bits of the state. This can be seen in the implementation as *xsub[0]*, which holds the LSB bits of the state, isn't returned to the user. This can be seen in the following diagram.



Entropy use: this implementation **doesn't** add any entropy to the PRNG.

Period: the implementation reaches the maximal period of $\mathbf{2^{48}}$ (size of m).

State: the state size is **48 bits**.

Seed: there is an option to set the seed externally. The seed setting alternatives follow.

1. ***srand48*** – Using this function we can supply **32 bits** seed. The initialization takes the 32 bits supplied and sets the 32 MSB bits of the state. The 16 LSB bits of the state are set to a **constant of 0x330E**. The resulting state is:

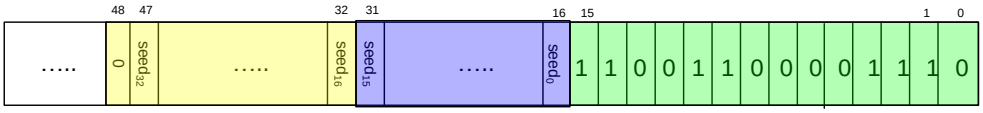


Figure 14 The State after Initialization Using srand48

2. ***seed48*** – Using this function the user can supply 48 bits of seed which will be translated to the initial state of the generator.
3. ***lcong48*** – Using this function the user can control all the properties of the generator: she can set the initial state and control the values of c and *a* of the LCG.

Default seed implementation: there **isn't any default seed**. Following this, calling any of the functions of this generator without setting the seed would result in the returned sequence as if **0** was the initial seed, meaning the fixed value of 0. From a developer point of view this decision has a disadvantage as it burdens the developer (who isn't always knowledgeable in the disadvantages of using a weak seed) with the responsibility of giving a strong enough seed to the generator.

5.5.4 Properties Analysis

Since this implementation is identical to one that we covered in 6.3 the reader is encouraged to view the analysis carried there. The only difference is that in this implementation we don't have a default seed, thus the attack on the seed proposed there isn't applicable here.

6 Java

6.1 Introduction

The following details were extracted from Java SDK (Software Development Kit) version 6u1 [76] dated 29/3/2007.

There are 3 major flavors of PRNG implementations in the JDK (Java Development Kit). Two of which, *Math.Random* and *java.util.Random,* are the same implementation with a different API. The third one, *java.security.SecureRandom*, is a complicated implementation, designed to be the secure PRNG to be used in security sensitive applications. This flavor also has several modes of operation, which are configuration and operating system dependent.

6.2 Math.Random

http://download.oracle.com/javase/6/docs/api/java/lang/Math.html

6.2.1 Design Space

API: Math.Random has a public method called *random()* whose implementation follows:

```
1    if (randomNumberGenerator == null) initRNG();
2    return randomNumberGenerator.nextDouble();
```

Figure 15 Math.Random random method code

The method returns a `double` value with a positive sign, greater than or equal to `0.0` and less than `1.0`. Returned values are chosen pseudo randomly with (approximately) uniform distribution from that range (from API doc).

The class holds a private static *java.util.Random* object named *randomNumberGenerator*. This object is initialized on the first call of *random()*. The initialization is done with the default seed, meaning calling the default constructor of *java.util.Random*. (See analysis of java.util.Random in 6.3.3 for default seed)

This *random()* method is only a wrapper method for the *java.util.Random.nextDouble()* method. Given this fact we will not discuss the PRNG issues for this flavor, since it is covered in details in the next section (java.util.Random).

6.3 java.util.Random

http://download-llnw.oracle.com/javase/6/docs/api/java/security/SecureRandom.html

6.3.1 Design Space

The analysis is based on version 1.47 of *java.util.Random*.

The API of *java.util.Random* is comprised of the following methods:

```
1    synchronized public void setSeed(long seed) { ... }
2    protected int next(int bits) { ... }
3    public void nextBytes(byte[] bytes) { ... }
4    public int nextInt() { ... }
5    public int nextInt(int n) { ... }
6    public long nextLong() { ... }
7    public boolean nextBoolean() { ... }
8    public float nextFloat() { ... }
9    public double nextDouble() { ... }
10   synchronized public double nextGaussian() { ... }
```

Figure 16 java.util.Random API methods

All the methods finally invoke the main *next(int bits)* method which steps the generator.

6.3.2 Under the Hood

The theoretical PRNG behind it is the LCG PRNG: more specifically it uses LCG's implementation as introduced in [1], which was also analyzed in the *rand48* functions family (see section 5.5).

The Java LCG's implementation code is as follows:

```
1    protected int next(int bits) {
2      seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1)
3      return (int)(seed >>> (48 - bits))
4    }
```

The recursion LCG formula is:

$$X_{n+1}=(a*X_n + c) \bmod m,\ n>=0,$$

$a = (25214903917)_{10}$
$c = (11)_{10}$
$m = 2^{48}$

Parameters choice: this choice of (a, c, m) guarantee an LCG with a maximal cycle, as shown in 5.5.3.

m: m is chosen as a power of 2, this means the implementation can be with the '&' operator, thus gaining performance. However, as described in [1] this results in a shorter cycle of the low order bits of the state than of the state as a whole. Probably due to this, the implementers chose to take only the upper 48 bits.

The output of the generator is **truncated** by shifting the state right (unsigned shift >>>) **by (*48-bits*) bits**, where *bits* is the amount of bits requested by the calling method. E.g., the method *nextInt* invokes *next(32)* in order to get an integer value.

Entropy use: This implementation **doesn't** add any entropy to the PRNG.

Period: the implementation reaches the maximal cycle of the LCG: $\mathbf{2^{48}}$ (the size of m).

State: The state size of the PRNG is **48 bits** due to the modulus used.

Seed: There is an option to set a seed externally and there's a default implementation for a seed. The seed is represented by a 64 bit integer, of which only the 48 LSb bits are used. This can be seen in the following code snippet.

```
1    (seed ^ 0x5DEECE66DL) & ((1L << 48) - 1)
```

Default seed implementation: There is an implementation for obtaining a default seed. The implementation is based upon obtaining a timestamp as can be seen in the following pseudo-code:

```
1    seed = (SU++) + current timestamp_nano
2    seed = (seed⊕a)&((1L << 48) - 1)
```

Figure 17 java.util.Random default seed implementation

Where the initial value of constant SU = 8682522807148012.

Nano seconds precision: The default seed implementation gets the current time with Nano seconds precision. The value is platform depended. The Java doc only states it's a time with Nano seconds precision from a certain constant time (not necessary the "epoch" of 1970). Moreover the value can be negative.

Seed Uniquifier (SU): in order to make consequent creations of default Random objects create different Random streams, a static long is added to the default seed. This value progresses by 1 each time a default Random object is created. We note that the choice for the specific initial value of parameter *SU* is not documented, nor does it seem to affect the security of the generator.

6.3.3 Properties Analysis

6.3.3.1 Pseudo-randomness

Assuming we know the implementation is based on *java.util.Random*: This will give us the information that the implementation is LCG based and we also know the parameters of this LCG.

Known cipher text attack (Finding the missing bits from the state): In this attack we assume that we get one or more outputs from the generator. If the generator was a classic LCG implementation, meaning the output was the state it-self and not a truncated version of the state, we would have had the state in our hands. However, since there isn't any public method that outputs the entire state we need to find those truncated missing bits.

Reminding that when *n* random bits are requested, the generator returns the *n* MSb from the state, meaning (*48-n*) bits are chucked from the state before returning the output; the attacker needs to break a space of 2^{48-n}.

Under the assumption that most implementations would require getting **random numbers (integers)** an attacker has to break a space of $\mathbf{2^{16}}$ in order to reveal the entire state, which is feasible. We still need a way to verify our guess.

Denote *H* as the 32 bit we get from the output and *L* as the 16 bits that we need to guess. Thus, the state at step *i*, S_i, can be written as:

$$S_i = 2^{16}H_i + L_i\ , |S| = 48\ bits$$

Number of outputs needed to verify our guess: Considering LCG as a random function, the expected number of different *L'* (besides the true value) that gets us to the same *H* as our guess is $2^{-32}(2^{16} - 1) \cong 2^{-16}$, so 1 output should be enough to validate the missing bits of the seed.

Note that this attack is still valid if we were to know how many times the generator was stepped before getting the second output.

Now, let's assume we **don't know we are dealing with *java.util.Random***. If we get consecutives outputs of 32 bits each, since the attack above is efficient we just need to mount the attack and finish.

6.3.3.2 Backward Security

None (not entropy based).

6.3.3.3 Forward Security

None; since it uses LCG, with knowledge of the current state we can simply reverse the LCG and get to the previous states.

6.3.3.4 Default Seed Weakness

The role of SU: since SU is a simple sequential number - we can try all the sequences after reducing the attack on the time based portion of the seed.

Now, let's review **how many bits of entropy** there are actually in the default seed: usually programs construct new *Random* objects in two scenarios (a) at the application startup as static instance and share it between different consumers; (b) as new objects each time a random value is needed.

(a) Application startup – knowing the exact application startup can be hard, however some assumptions can be done. We can assume that the application startup is the same as the server uptime (in most servers the application will be a system service that would startup shortly

after the server's startup). In the worst case let's assume we know nothing about the actual server's uptime: We can assume that the server goes offline once a year since even a 5 nines of availability states 5 minutes of downtime per year, this reduces our attack space to $log_2(365_{days} * 24_{hrs} * 60_{mins} * 60_{sec} * 1000_{ms} * 1{,}000{,}000_{ns)} = 54.80 \sim= \mathbf{55\ bits}$.

We can **further improve** our attack if we were to allow some more assumptions; if the server is open to an ***nmap*** probe[3] or to other techniques, such as packet sequencing guessing, then the server's uptime can be pin pointed to the exact second. Here's a typical *nmap* output regarding the uptime of a server:

11.236 days (since Wed Oct 28 14:01:57 2009)

This can reduce our attack to: $log_2(1_{sec} * 1000_{ms} * 1000000_{ns}) = 29.89 \sim= \mathbf{30\ bits}$.

An even less aggressive assumption can be made by looking at some statistics regarding *average server uptime* values. Such values are available at sites such as http://uptime.netcraft.com.

(b) New objects for each invocation – knowing the current time of the attacked server can be found by many techniques, such as observing the *Hello* message in the SSL protocol handshake [77]. Even if these methods aren't applicable we can assume that most servers today synchronize their time somehow (via NTP[4] or other measures), thus a good assumption is that the attacker can guess the current time with a **minute** of error. This reduces the attack to only

$$log_2 (1_{min} * 60_{sec} * 1000_{ms} * 1000000_{ns}) = 35.804 \sim= \mathbf{36\ bits}.$$ [5]

[3] http://nmap.org/

[4] http://www.ntp.org/

[5] Actually, according to the JVM implementation on Windows, this could be even weaker. If the Windows machine doesn't support *High Performance Counters*, then the implementation falls back to: Timems * 1,000,000. This actually results in no more than milliseconds precision, which leads to $log_2(1 * 60 * 1000) = 15.87\ bits = \mathbf{16}\ bits$

6.4 java.security.SecureRandom

http://download-llnw.oracle.com/javase/6/docs/api/java/security/SecureRandom.html

6.4.1 Introduction

This is the stronger implementation of a PRNG that exists in Java. As stated in the documentation *"… this implementation **minimally** complies with the statistical random number generator tests specified in FIPS 140-2* [78]*, Security Requirements for Cryptographic Modules, section 4.9.1. Additionally, SecureRandom must produce non-deterministic output. Therefore any seed material passed to a SecureRandom object must be unpredictable, and all SecureRandom output sequences must be cryptographically strong, as described in RFC 1750* [79]*: Randomness Recommendations for Security"*.

This section is organized in the following manner: The design space section surveys the general architecture and design of the SecureRandom providers. We then analyze each of the various providers in respect to the specific design space and prng algorithm used. We then follow to discuss the PRNG analysis of each provider.

6.4.2 Design Space

The analysis is based on version 1.54 of *java.security.SecureRandom*.

As given by its name, Sun states that this flavor of prng is a **secure** one. It relies on JCA (Java Cryptographic Architecture), thus allowing pluggable JCA providers to be added and used (e.g., on Windows platforms one can use the Windows-PRNG as exposed by MS CryptoAPI (CAPI)). The **default PRNG** provider is Sun's provider, which utilizes SHA-1 as the PRNG algorithm.

For an elaborate explanation of JCA refer to the documentation in [80].

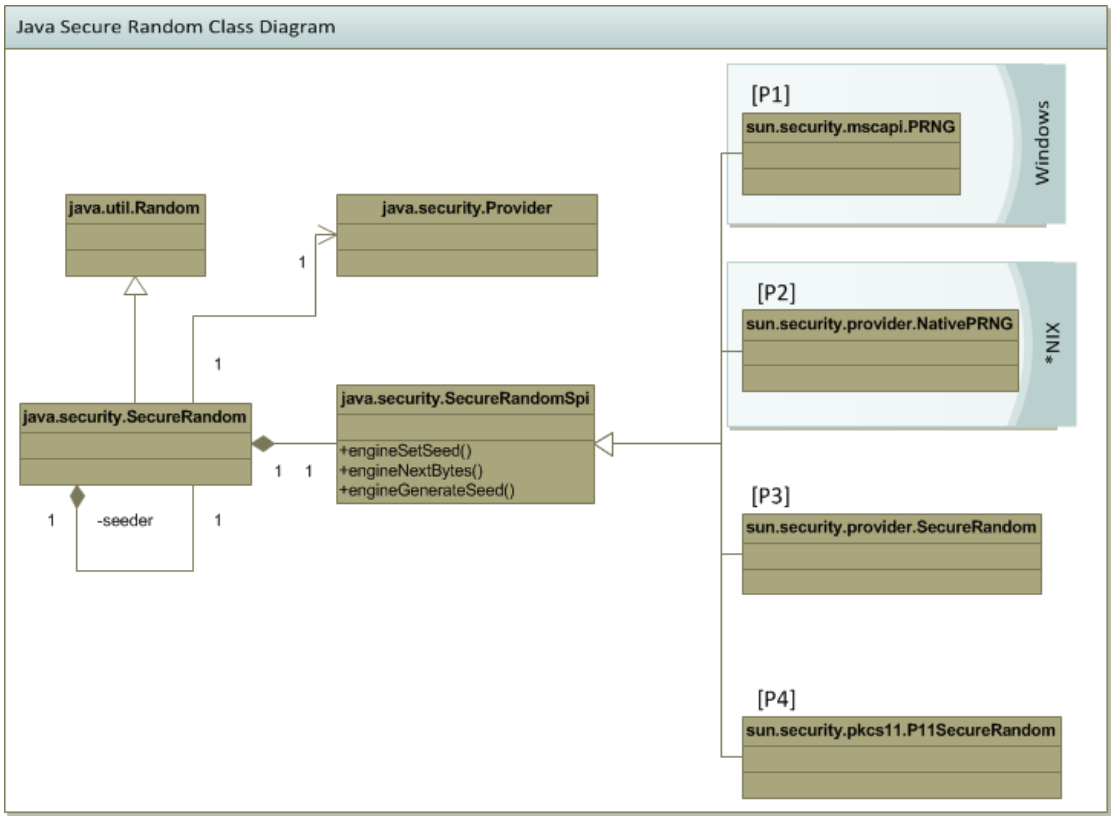


Figure 18 SecureRandom Class Diagram of Default Available SecureRandomSpi

Figure 18 shows the class diagram of the various SecureRandom providers that are available in the JDK. The *java.security.SecureRandom* extends **java.util.Random** and uses the resolved implementer ***java.security.SecureRandomSpi,*** which is the *SecureRandom* Service Provider Interface, in order to perform the actual work.

Provider API: The Service Provider, ***java.security.SecureRandomSpi,*** dictates the following operations:

```
1    protected abstract void engineSetSeed(byte[] seed);
2    protected abstract void engineNextBytes(byte[] bytes);
3    protected abstract byte[] engineGenerateSeed(int numBytes);
```

Figure 19 SecureRandomSpi API methods

Besides the obvious operations of setting the seed (*engineSetSeed)* and getting random bytes from the engine (*engineNextBytes*), each specification needs to also support generating a seed (*engineGenerateSeed*). **This is a notable design decision** that allows using different implementations for generating seeds and asserting the semantics that a seed should be treated differently than just getting bytes from the generator.

Resolving: The resolving of the appropriate Provider is implemented in the ***getPrngAlgorithm*** method as follows: it finds the **first** available JCA provider that implements the "SecureRandom" service. Since the JCA contract for providers state that the providers are placed in an array in a 1-based order of precedence, the first provider has the highest precedence, thus the "best" provider is chosen. The provider provides all the JCA services and not necessarily the implementation of the random generator.[6] If no provider implements the "SecureRandom" service the default Sun's provider is chosen.

java.security **configuration file**: the configuration and resolving of providers (the ordering and priority of providers) is written in the *java.security* configuration file. The default contents of the configuration file, as shipped with new JDK installations can be seen in 12.1. The file is shipped with every installation of JDK/JRE of java under the *lib/security* folder.

The fact that the file resides in the file-system and without any protection can contribute to a downgrading attack. Assuming the attacker has access to the folder in which the JRE/JDK is installed, the attacker can change the order in which the providers are defined and to cause a less secure PRNG to be invoked. Naturally, this depends on the permission of the installation directories, in which the user installed the JRE. However Sun could have put it in a more secure location, such as, on Windows platforms, the registry location to prevent this kind of error. We note that even if the file is only readable, knowing the chosen Provider is information that can benefit the attacker.

Delegation: After the Provider is chosen, all the *Random* methods simply delegate the calls to the providers` relevant methods.

Implementations: there are several implementations for *SecureRandomSpi* that are available in JDK. Below is a summary of the available implementations and applicable platforms; this can also be seen in Figure 18.

[6] This means that if, for instance, we have a great SecureRandom provider but a really bad RSA or DSA implementation it will be picked also for the RSA/DSA implementation. This is due to the fact that the order of precedence of the providers in JCA is general and not service-dependent.

#	Implementation	Platform
P1	sun.security.mscapi.PRNG	Windows
P2	sun.security.provider.nativePRNG	*NIX (Linux/Solaris)
P3	Sun.security.provider.SecureRandom	Independent
P4	sun.security.provider.pkcs11.P11SecureRandom	N/A (PKCS)

Table 2SecureRandomSpi implementations

We now continue and examine each one of the above implementations.

6.4.3 P1: MSCapi PRNG

6.4.3.1 Design Space

The implementation utilizes the MS-CAPI infrastructure [81]. The implementation is a native implementation that uses the ***CryptGenRandom*** API method to produce random bits.

Implementation: The implementation can be reviewed in the JDK source files under **\src\j2se\src\windows\native\sun\security\mscapi\security.cpp.** All of the API methods are delegated to one native function - ***generatedSeed***:

```
1    native byte[] generateSeed(int length, byte[] seed);
```

The different ***SecureRandomSpi*** methods are implemented using a convention of the *length* parameter. (a) If length < 0: the implementation uses the supplied seed to re-seed the generator, (b) if length = 0: the implementation generates new random bytes of *sizeof(seed)* and places it in the *seed* parameter, (c) if length > 0: the implementation generates new random bytes of specified *length*.

6.4.3.2 Under the Hood and Properties Analysis

The implementation uses the default implementation that exists in the MS-CAPI implementation, including the default seed generation.

For a detailed analysis of the generator that is used and its security properties in the MS-CAPI infrastructure please refer to the analysis of the Windows Random Number Generator in 3.8.

6.4.4 P2: nativePRNG

6.4.4.1 Design Space

This implementation utilizes */dev/random* and */dev/urandom*. This is the OS based generator that is available in *NIX. Please refer to 3.7 for an explanation of the algorithm used by this generator.

The implementation uses a singleton instance of the inner class ***RandomIO*** for the actual work. According to the comments, this is in order to not open the file descriptors of */dev/[u]random* every time. The singleton instance is instantiated during class initialization.

Urandom buffer (urandom_buf): The implementation uses a buffer on top of */dev/urandom* of **32 bytes**. The buffer also has a **freshness time, T_{fresh},** of **100 msec**. This means that we will always reset the buffer when trying to read data which passed this freshness time. As explained in the actual code comments this is to prevent the implementation from reading stale data from the buffer.

Mixing algorithm: The implementation also uses the default SUN implementation, ***P3***, in order to mix in random bytes from both implementations. According to the documentation, this instance is used in tandem to the native generator, in order to make sure the user has the ability to set an

external seed. This seems like a technical/design decision, since setting an external seed under *NIX platforms must be performed by writing to the ***/dev/[u]random*** device and in some systems it requires root access. The designers chose to add the P3 instance in order to fully support the SecureRandom API, which requires the ability to seed an external seed. **All** invocations of this flavor also mix random bytes as read from ***P3***.

Initialization process: the initialization process opens the input streams from both devices and initializes the urandom buffer. The initialization of the mixing algorithm is lazy, upon demand. It is initialized by reading 20 bytes **directly** from the **/dev/urandom** input stream.

6.4.4.2 Under The Hood

PRNG implementation: the algorithm implementation gets random bytes from the *urandom buffer* and XORs it with random bytes from *P3* (as its mixing algorithm). The pseudo code for getting bytes from the generator follows.

```
1     if P3 is NOT initialized then       // initialize P3
2           read(/dev/urandom, buf);
3           P3.engineSetSeed(buf);        // buf is 20 bytes in size
4     end
5
6     // put random bytes from P3 in outBuf
7     P3.engineNextBytes(outBuf);
8
9
10    // Get data from the urandom_buffer (fill if needed) and XOR with P3
11    len = outBuf.length;
12    offset = 0;
13    while (len > 0) Do
14          fill_urandom_buf_if_needed(); // Check if we need to fill
                                          urandom_buf
15
16          outBuf[offset] ^= urandom_buf[offset];
17          len--; offset++;
18    end
```

Figure 20 engineNextBytes(byte[] outBuf) pseudo code

Adding data to the urandom buffer (fill_urandom_buf_if_needed): the implementation always returns data to the caller from the urandom buffer. In case the buffer is empty or the data in it exceeded T_{fresh} the buffer is filled by reading a chunk of 32 bytes from /dev/urandom.

Setting an external seed: an external seed is set by trying to write to the /dev/random device **and** by setting the seed in the mixing algorithm *P3*. In case */dev/random* is not available for writing the seed will only be set in the mixing algorithm. We note that the external seed doesn't **replace** the state in any of the generators, but is added to the entropy pools (see [49] and 3.7 for details). The pseudo code for setting the seed follows.

```
1     if (/dev/random accessible) then
2           write to /dev/random seed bytes.
3     end
4     P3.engineSetSeed(seed);
```

Figure 21 engineSetSeed(byte[] seed) pseudo code

Generating a seed: we remind ourselves that each provider should support an API method of generating a seed. The implementation reads the specified *numBytes* from */dev/random* in order to generate a new seed.

Default seed: the implementation lets the LRNG use its own default seed and doesn't include any alternative default seed on its own. The *P3* generator is seeded by reading 20 bytes from the ***/dev/urandom*** interface.

/dev/urandom vs. /dev/random: the major difference between the two devices, as explained in 3.7, is that the /dev/random device blocks if there isn't enough entropy in its pool. By having this property, the /dev/random output is considered "extremely strong"[7] as it always uses more entropy. /dev/urandom implementation is the non-blocking device. Here these two devices are used in tandem: /dev/urandom as the main device being used in order to produce the actual random bytes for consumers and /dev/random only if this generator is used in order to generate a seed.

Entropy Use: by using the random devices available on *NIX platforms this implementation utilizes entropy from various inputs (such as input devices, hard disk writes etc.). For details regarding the exact entropy extracting algorithm the user is encouraged to read [49].

State: the state of this generator is dictated by the states of both generators:

1. The size of the primary entropy pool that is used by /dev/[u]random devices. As seen in 3.7 the size of the pool is 512 bytes. If we'll allow ourselves to examine only the state size of the *urandom* pool, we get a size of 128 bytes=**1024 bits.**
2. The size of the state of *P3* is 20 bytes, which is **160** bits.

This leads us to a combined length of **1184** bits.

Period: the algorithm used relies on various entropy sources and utilizes several invocation of a SHA-1 variant. Due to the fact that the LRNG uses an external source of entropy, discussing the period for this implementation is irrelevant.

6.4.4.3 Properties Analysis

6.4.4.3.1 Pseudo-randomness

Assuming we know the implementation is *P2*.

Brute Force: A brute force attack requires going over the entire space of one of the pools that are used. Taking the 32 words pool of */dev/urandom* it will require $\mathbf{2^{1024}}$, which is not a feasible attack space. We also have to take under consideration the *P3* generator, of which a brute force would require breaking another $\mathbf{2^{160}}$ space. This takes us to a combined strength of $\mathbf{2^{1184}}$. Here we assumed that *P3's* output is independent of the LRNG.

Entropy use: we note that due to the entropy added to the generator, brute force is only applicable during times that no entropy is added to the states.

Dependence of the LRNG and P3's seed: since P3 is initialized from the /dev/urandom and then XORed with bytes from urandom_buf in order to get outputs might lead to weaker effective entropy of the whole construct due to both using outputs from /dev/urandom. Furthermore, the **composition of two generators**, LRNG and *P3*, is a choice that resulted from a design/deployment issue. Although we didn't find any weakness that resulted from these dependencies this doesn't necessarily make the scheme more secure.

The usage of /dev/urandom and not get_random_bytes: according to [82] (page 40, chapter 6.2) there is a weakness in environments that only use the device interfaces of the LRNG and don't use the kernel API function *get_random_bytes*. The main weakness is that the *urandom* pool is not refilled with entropy from the primary pool when the system only utilizes the devices in order to produce randomness. Although we couldn't think of a deployment that would **never** use *get_random_bytes*, our implementation fits the paper's concern as it only uses /dev/urandom in order to get random bits.

[7] These words were used by the designer of the linux random generator. For a detailed discussion regarding this, please refer to [49] chapter 3.4.

Performance vs. Security:

1. **Only using /dev/urandom:** the developers clearly state in the class documentation that the fact they **don't require** getting data from /dev/random as a positive attribute of this implementation from the fear of the blocking nature of /dev/random. For instance, this choice makes the implementation vulnerable to the weakness expressed in [49] in case the consuming rate is higher than the entropy generation rate. Although noting this is probably a reasonable decision, a better design would have also allowed the user to configure whether she prefers to use /dev/random for extra security, while risking blocking her application.
2. **Buffering of /dev/urandom and T_{fresh} as a hidden security parameter:** the reasoning for choosing 100ms as the freshness time isn't explained nor documented. We note that this introduces another security parameter that can directly contribute to the security of the scheme. Consider a situation where the buffered 32 bytes (or part of them) were read from /dev/urandom at a point that the entropy level was poor and immediately after reading these bytes, the entropy was refreshed. If our implementation wouldn't have performed the buffering and simply read from the device, this low entropy might have influenced only some bytes and not the entire 32 bytes.

6.4.4.3.2 Backwards Security

We are not aware of any attacks on the backward security of LRNG, so we can conclude that this implementation has a 2^{1184} backwards security strength as-well.

6.4.4.3.3 Forward Security

Forward security is achieved by having the entropy in the LRNG refreshed throughout the generator's work. As shown in 3.7, there is an attack on the forward security of LRNG with a time complexity of $\mathbf{2^{64}}$ *or* $\mathbf{2^{96}}$, depending on the attack variant while the memory complexity is $O(1)$.

We point out that the fact the generator is always XORed with a *P3* instance is irrelevant to the forward security of the construction. This is due to the fact that no entropy is added to *P3s* state during the generator's work.

6.4.4.3.4 Seed Security

LRNG seed: the seed implementation of the LRNG is based on various machine generated entropy. A thorough review of the entropy generator mechanism and its weaknesses are presented in [49].

Usage of /dev/urandom for P3's seed: the fact that the seed for P3 is also from */dev/urandom* can also result in a weaker seed for P3 for reasons noted earlier. If the implementers already gone to the length of having this generator alongside the native one, a better entropy seed could have been achieved using /dev/random.

6.4.5 P4: P11SecureRandom – PKCS-11 implementation

An implementation for a PKCS-11 [83] environment; this implementation is out of scope for this work. The implementation is for environments that, for instance, use smart-cards; then these smart-cards are accessed via this implementation in order to produce random bits using the smart-cards hardware random generator.

6.4.6 P3: Sun's **default** PRNG implementation: SecureRandom

6.4.6.1 Design Space

The sun implementation can be viewed in the implementation class sun.security.provider.SecureRandom. This is the **default implementation** of the SecureRandom provider.

The implementation provides a platform independent implementation with the use of an underlying algorithm based on SHA-1 (see 6.4.6.2 for algorithm details).

In this implementation we are introduced for the first time to a new class, responsible for the seeding operation. The class is *sun.security.provider.**SeedGenerator*** and has only package level visibility, so only security provider implementations can use it.

The implementation uses the following members to implement the generator:

```
1    private static SecureRandom seeder;
2
3    private transient MessageDigest digest;
4
5    private byte[] state;
6    private byte[] remainder;
7    private int remCount;
```

seeder: a class member which holds the implementation for the seeding function. The seeder itself is another instance of the *SecureRandom* generator, which is initialized using the *SeedGenerator* implementation.

remainder: an array that holds the remainder of the state, as saved between intermediate operations of the generator. **remCount** is the actual size of the remainder.

state: the state of our generator, the state is updated in each needed run of the generator.

SeedGenerator: the addition of the *SeedGenerator* class makes the implementation of P3 even more cumbersome, as now we have another implementation which is only available for seeding purposes and has different implementation between platforms. This class has a set of resolving rules of its own in order to decide which seeding algorithm to use. The class diagram of the seed generator procedure follows.

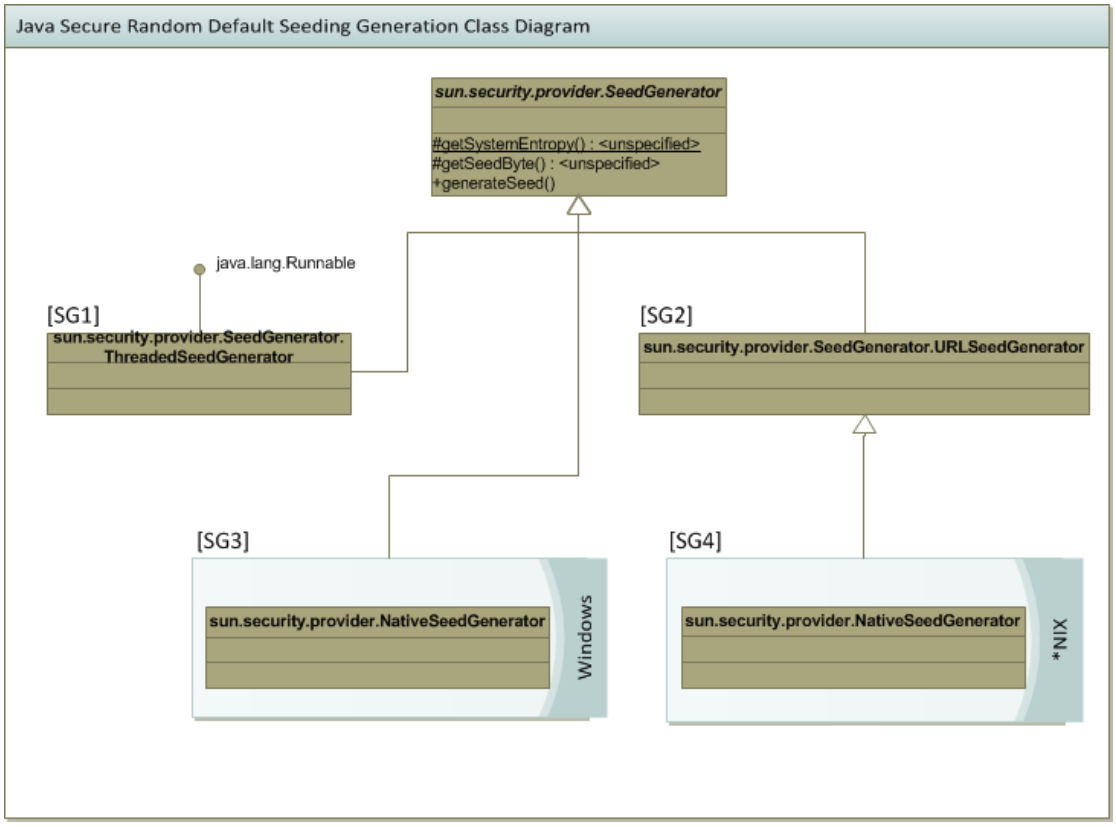


Figure 22 Seeding Generation Class Diagram

*sun.security.provider.**SeedGenerator*** has three concrete implementations:

1. ***SG1:*** *sun.security.provider.SeedGenerator. ThreadedSeedGenerator* – the fallback implementation provided by Sun. A platform **independent** implementation that utilizes various system events as entropy sources. More details of this implementation in 6.4.6.2.
2. ***SG2:*** *sun.security.provider.SeedGenerator.URLSeedGenerator* – an implementation that invokes a URL in order to receive random bytes from it. Can be useful, for instance, in order to use the *egd* (Entropy Gathering Daemon) [84] or services like www.random.org.
3. **SG4:** *sun.security.provider.NativeSeedGeneator (*NIX)* – shipped with UNIX variants. The class is implemented by extending the *URLSeedGenerator* for reading from the **file:/dev/random** interface as the URL.
4. ***SG3:*** *sun.security.provider.NativeSeedGeneator* (Windows) – a native seed generator, which is available only on Windows machines. Gets the seed from the MS-CAPI interface, *CryptGenRandom* function.

Seed Generator Resolving: the resolving of the proper seed generator is based on a configuration value that exists either in the *java.security* file or as a system property passed to the VM. In-case **either** */dev/random* or */dev/urandom* is passed as a value to the key *securerandom.source* the NativeSeedGenerator is used: On a Windows machine, the Windows variant of the NativeSeedGenerator is used and on a *NIX machine, the */dev/random* NativeSeedGenerator is used. We note that even if the user explicitly wants to use */dev/**u**random* as the source, the implementation still uses the blocking interface */dev/random*. It seems like a decision to guarantee the seed is strong enough.

SeedGenerator's **API methods:**

1. ***getSystemEntropy():*** implemented in the SeedGenerator class itself, produces System based entropy. Exact entropy sources and algorithm details exist in the next section.
2. ***generateSeed(byte[] result):*** used to generate a seed using the resolved seed generator by invoking the abstract method *getSeedByte()*.

6.4.6.2 Under the Hood

The theoretical PRNG behind: the PRNG uses SHA-1 invocation over the provider's state. The algorithm follows:

$$x_n = SHA1(s_n)$$

$$s_n = \begin{cases} initial\ seed & n = 0 \\ (x_{n-1} + s_{n-1} + 1) \bmod 2^{160} & n > 1 \end{cases}$$

Figure 23 Sun's default generator

Where X_n is the generator's next output and S_n is the generator's internal state.

Entropy use: there isn't any use of entropy to refresh the state of the generator.

Period: consider the function $S_n \rightarrow S_{n+1}$ as a random function, following the birthday paradox [85] we should expect an average cycle length of roughly 2^{80}.

State: the state is **20** bytes in size.

Seed: (option to set the seed externally): there is an option to externally set the seed. In case there is already a state to the generator, the seed doesn't replace it but only supplements on the current state. Let *ext_seed* be the external seed to be used; this is done by the following:

$$s_n = SHA1(s_{n-1}.concat(ext\_seed))$$

Generating a seed: reminding ourselves that each provider should support an API method of generating a seed, the implementation invokes *SeedGenerator.generateSeed()* in order to generate the seed for the caller.

(Default) Initial seed (S_0): the default seed generation process is lazy and performed upon demand, requiring the use of the seeder instance. We continue with a detailed description of this process. We use the following notations:

1. ***SG**** - the resolved SG (SeedGenerator), as resolved in the description above.
2. ***seeder*** – the instance of the seeding generator.
3. ***prng*** - *the* instance of the actual PRNG used to output the random bits.

```
1    // Step1 - Get system entropy - seed the seeder, |systemEntropy|=20B
2    systemEntropy = SeedGenerator.getSystemEntropy();
4    seeder_x0 = SHA1(systemEntropy)=SHA1(seeder_s0);
4
5    // Step2 - Get SeedGenerator dependent random data, |SG*Output|=20B
6    SG*Output = SG*.generateSeed(20 bytes);
7
8    // Reseed the seeder with SG*Output
9    seeder_x0 = SHA1(seeder_x0.concat(SG*Output));
10
11   // Step the seeder state and get the initial seed for the algorithm
12   prng_s0  = seeder_x1 = SHA1(seeder_x0)
```

Figure 24 P3 default seed algorithm

The **system entropy** is collected from 3 major entropy sources – time based entropy, system configuration and system runtime state:

1. **Time** based entropy – taking one byte of the current-time, in msec.
2. **System configuration**:
    a. Names and values of the JVM system properties.
    b. The name and IP address of the machine.
3. **System runtime** state:
    a. The filenames of the *java.io.tmpdir* folder.
    b. Snapshot of memory state of the JVM: total memory allocated and the free memory allocated for the VM.

The algorithm for generating the system entropy follows:

```
1    // Time consideration
2    // 64 bits of time since epoch (1/1/1970) → 8 LSb bits
3    lsbTimeByte = (byte) System.currentTimeMillis();
4    buffer.concat(lsbTimeByte);
5
6    // System properties
7    For (property in System.getProperties()) Do
8      buffer.concat(property.getName());
9      buffer.concat(property.getValue());
10   end
11
12   // Computer name and IP
13   buffer.concat(ipAddress + '/' + localMachineName);
14
15   // Filenames from the TEMP (java.io.tmpdir) directory
16   For (fileName in files(java.io.tmpdir)) Do
17     buffer.concat(fileName);
18   end
19
20   // Memory stats
21   buffer.concat(totalMemoryLength); // The total VM memory
22   buffer.concat(freeMemoryLength);  // The free VM memory
```

```
23
24    system_entropy=SHA1(buffer);
```

Figure 25 P3 system entropy gathering

We continue to describe the implementation details of the **SG implementations:**

1. ***SG2:*** *sun.security.provider.SeedGenerator.URLSeedGenerator* – the implementation opens a **BufferedOutputStream** with the default buffer size of **8 KB** to the supplied URL.
2. **SG4**: *sun.security.provider.NativeSeedGeneator (*NIX)* – implemented by extending the *URLSeedGenerator* with the URL **file:/dev/random**. Source file can be found in the JDK source archive, under j2se\src\solaris\classes\sun\security\provider.
3. **SG3**: sun.security.provider.NativeSeedGeneator (Windows) – the implementation invokes a native method named *nativeGenerateSeed.* The method is natively implemented using the WRNG (see 3.8) by invoking ***CryptGenRandom****.* Source file can be found in the JDK source archive under j2se\src\windows\classes\sun\security\provider and the native implementation is under j2se\src\windows\native\sun\security\provider.
4. ***SG1:*** *sun.security.provider.SeedGenerator. ThreadedSeedGenerator* – the *SG1* seeder is only used as a **fallback mechanism** for installations where no native support for OS based PRNG is available (such as Solaris of version < 10). The implementation of SG1 is quite complicated and it seems to be well thought off. Its description follows.

SG1 detailed description: at the core of the implementation there is a thread that is responsible to keep a random bytes *queue*, denote ***entropy_queue***. The implementation uses this thread to asynchronously fill the random bytes queue. Random bytes are returned to the caller by simply getting bytes from the queue. In case the queue is empty the caller waits for the queue to fill up again.

Performance of SG1: we note that the initialization process of this thread, from a performance point of view is very bad. The first SecureRandom instance that is instantiated blocks in order to create this thread, until sufficient entropy is in the queue. On a standard PC it took **several seconds** to instantiate this thread.

Entropy gathering in SG1: the process gathers entropy by trying to estimate the load on the machine. The following pseudo-code describes this process.

```
1    While (counter < 64,000 and quanta < 6) Do
2      // Spawn a bogus-thread
3      new BogusThread().start();
4
5      // How many operations can we perform in 250 msec?
6      For (250 msec) Do
7            syncrhonized (this) {};
8            numOperations++;
9      end
10
11     value = value XOR perm_table[numOperations % 255]; // |value|=1B
12     counter += numOperations;
13     quanta++;
14   end
15
16   queue.push(value); // Add the generated entropy byte to the queue
```

Figure 26 SG1 entropy gathering algorithm

Whereas ***BogusThread*** is a thread that is spawned in order to add some entropy to the thread scheduler. The *BogusThread* implementation sleeps for 250ms by iterating 5 times and sleeping for 50ms in each iteration.

We couldn't find any reasoning as to why the implementation uses the above parameters, such as the 250ms and spin-count of 64,000.

perm_table is a fixed permutation table of 255 values that can be seen in 11.1.1.1. According to the code comments, the table was generated by generating 64k of random data and using it to mix the *Trivial Permutation*.

6.4.6.3 Properties Analysis

6.4.6.3.1 Pseudo-randomness

By using the *SHA1* as a primary building block we couldn't find a better attack than the actual size of the *SHA1* state buffer, which is 20 bytes. We can conclude that the strength of this variant is $\mathbf{2^{160}}$strong.

Proprietary implementation: the generator that Sun chose to have as the default generator is a proprietary implementation that although the use of known building blocks is still utilized in a proprietary way. The seeding generation algorithm, according to the documentation of sun.security.provider.SecureRandom, was never thoroughly reviewed nor widely deployed (this can be seen in the java-doc of the default constructor of *SecureRandom*).

6.4.6.3.2 Backward Security

None (not entropy based).

6.4.6.3.3 Forward Security

By using SHA1 as the core building block, we actually achieve a forward security that is the size of the SHA1 output, which is $\mathbf{2^{160}}$. We note that recent years SHA-1 collisions that were presented in [86] are actually not applicable for this scenario, since we want to **find** a previous state (or some of it) using a current state.

6.4.6.3.4 Default Seed Security

The default seed generation mechanism is one of the most complicated steps in this generator's implementation. The implementers went through great length to have enough entropy used in the default seed generation. In this section we'll note several vulnerabilities in some of the entropy sources used.

A rough estimation of the entropy we get from **Step1** (system entropy) of the seed generation follows.

1. **Current time** – since we only take the lower 8 bits of the current time, it is not easy to guess this value. This is due to the fact that 8 bits can represent 256 msec, which is a harsh requirement of the attacker to know the exact time of the time the seed was generated. Amount of expected entropy bits: 8 bits.
2. **System Configuration**:
    a. **Java system properties** – this source has very limited entropy as it is predictable between installations and machines. We've tested 4 machines evaluating this. The result is that per system environment configuration used (Windows or Linux and JRE version) the system properties were **completely identical**.
    We note that some application servers and popular java libraries, such as log4j, do support configuration via the java system properties mechanism.
    In this scenario it is still easy to predict the names and values from these configurations. E.g., consider a logging configuration filename for *log4:* we already know the key of the property to be *log4j.configuration*. The value is also easily guessable by trying popular names such as *log4j.xml*, *log_config.xml* etc.

We further note that in modern system most configuration are based on external configuration files, for better management and readability, so the actual use of the java system properties mechanism is not widely accepted.
Amount of expected entropy bits: ~0 bits.

b. **Name and IP address of the machine** – we can assume that the attacker has this information. This information is relatively easy to get by using various tools, such as *nmap*.
Amount of expected entropy bits: 0 bits.

3. **System runtime state**:
   a. The filenames of the ***java.io.tmp*** directory – this source is harder to predict as it requires a very intimate knowledge with the specific system, deployment and applications running on the target machine. We found it hard to actually limit the entropy inputs here. We note that in most environments periodical cleaning of the temp directory is performed. This leads to the fact that this entropy source is sensitive during times when the cleaning procedure had just been performed.
   b. **Memory size snapshot** – assuming that most attackers will target an enterprise grade application; we can assume that the changes in available memory aren't big. This is due to the fact that the pattern of enterprise software is to occupy a lot of the memory allocated to the JVM for application level caches. We can assume that changes are within a few hundred MB for an application that consumes 1-2GB. This gets us to amount of entropy of ~37 bits (100 MB) and ~74 bits (200 MB).
   Amount of expected entropy bits: ~37-74 bits.

We observe that if we were to allow ourselves to have access to the attacked machines, all the entropy sources above would have 0 entropy bits, as it is easy to know every parameter.

We note that the system entropy gathering, despite having very limited entropy which is of less than the desired 160 bit entropy is not as important for the seed strength since **Step2** of the seed generation can add entropy from native based generators.

Step2 of the seed generation depends on which *SG* was resolved during the SG resolving process. In case one of the native variants (*SG2*-*SG4*) of the *SG* was chosen, we don't have a weakness to share and thus we can assume that the strength of this step is the required $\mathbf{2^{160}}$. We note that the weakness described in 6.4.4.3.1 due to the use of /dev/urandom isn't applicable here, as the seed uses /dev/random.

SG1 Weaknesses: Since on the two major platforms, Linux and Windows, we don't fall back to the weaker seed generator *SG1* an analysis of the strength of *SG1* seems less important. We have tried to empirically perform tests regarding the amount of entropy we get from running the entropy collection procedure in SG1. Our results were quite surprising, as it seems that this procedure does achieve good entropy results. The fact that numOperations count is projected on 255 values, using the LSb values seem to remove dependence that naturally exists in the higher bits of the counter.

Controlling system load by an outside attacker: We state an obvious **weakness of the algorithm used for *SG1***. The attempt of using timing based entropy that correlate to the JVM load seems very risky. Consider an attacker who can control the perceived load of a machine. Then this attacker can attempt to produce a heavy load on the application, thus controlling the amount of entropy that the seed generator, *SG1* produces.

7 C# (.NET)

7.1 Introduction

The following details were extracted from Microsoft's implementation of .NET's **2** CLI [87], called CLR [57].

Despite the fact that Microsofts' CLR implementation is basically a closed source, Microsoft released to the general public a copy of the CLR source code, named Shared Source CLI (SSCLI) [88] dated 23/3/2006. We used this source code in order to understand the exact implementation of the *System.Security.Cryptography.RNGCryptoServiceProvider* generator. The source in SSCLI was verified to be the one that is used in the CLI version we covered using the tool .NET reflector [89]. Since we show in the following analysis a bug in the implementation of *System.Random* we also made sure that this bug also exists in *CLI* versions 3.5 (current) and 4 (a release candidate of the next .NET framework).

In order to verify the code path of the native code used in *RNGCryptoServiceProvider* we used the IDA-Pro [4] disassembler.

Giving the fact that most .NET applications are used on Microsoft Windows platforms we will review the implementation only for Microsoft's Windows platforms.

7.2 System.Random

http://msdn.microsoft.com/en-us/library/system.random.aspx

7.2.1 Design Space

The analysis is based on version 2.0.0.0 of *System.Random*.

The API of *System.Random* follows.

```
1    public Random();
2    public Random(int Seed);
3    public virtual int Next();
4    public virtual int Next(int maxValue);
5    public virtual int Next(int minValue, int maxValue);
6    public virtual void NextBytes(byte[] buffer);
7    public virtual double NextDouble();
8    protected virtual double Sample();
```

Figure 27 System.Random API

The actual code, as generated by the Reflector can be viewed in 11.2.1. All methods above invoke the main *internalSample()* method, which returns an *int* from the generator's internal state.

Ability to reset the seed during operation: unlike its Java counterpart, there isn't any API method to reset the seed after the Random instance had been created.

Absent of seed Uniquifier: consequent creation of *Random* objects can result in having the same random stream. This means that if different random sequences are important for the application care should be taken when creating multiple Random objects. The MSDN documentation recommends on creating only one Random object and getting random values from it. Another way to solve this is to add a unique sequence to each *Random* creation, or some other form of salt. We note that from a practical point of view most developers would simply create new *Random* objects without thinking/knowing about this issue. Java, on the other hand, as covered in 6.3.2, solved this in its implementation by adding the seedUniquifier.

7.2.2 Under the Hood

The theoretical PRNG behind it is the subtractive random number generator algorithm that was introduced by Knuth in [1]. The general algorithm is detailed in 2.4.5; however we'll repeat specific details here that deal with the implementation in C#.

The recurrence formula of the generator follows, whereas the parameters, using the notations that were used in writing the general algorithm in 2.4.5, here are *j=55* and *k=34*.

$$X_n = (X_{n-55} - X_{n-34}) \; mod \; (2^{31} - 1), \quad n \geq 0$$

The algorithm is implemented by keeping a circular list of 56 random numbers, which is initially filled as ma[1]=X_{55}, ma[2]=X_{54},…, ma[55]=X_1. The **initialization** is the process of filling the list from a seed and then randomizing the list using a deterministic algorithm. The implementation holds two pointers to the list, *inextp* and *inext*, which are kept **21** indices apart. **An output of a random integer** is the product of subtracting the two list values at the two pointers. This new random integer is also stored in the list.

Similarity to numerical recipes in C implementation: The implementation used here seems almost identical to one introduced in [2] *ran3* function. The main change that exists in this implementation is that instead of keeping the two indices **31** places apart, like Knuth suggests and like *ran3* is implemented, here the indices are **21** places apart. We note that the open source implementation of the CLR, the *mono* project [90] does implement this with the **31** value. A detailed analysis of the effects of this change is in 7.2.3.1 below.

Algorithm implementation: The detailed implementation in pseudo code follows.

1. Generator state:
    a. *ma*[56] – the array of 56 integers, where only indices 1..56 are actually used, thus we have 55 random numbers.
    b. *inext, inextp* – the pointers to the array.
2. Parameters:
    a. MBIG= 2147483647 = $(2^{31} - 1)$.
    b. MSEED= 161803398 (the "golden ratio" [91]).
    c. Seed – an integer seed to use in initialization.

Initialization process: First, we show how the algorithm initializes the generators` state:

```
1    tmp1 = MSEED - ABS(Seed);
2    ma[55] = tmp1;
3    tmp2 = 1;
4
5    For i in 1..54 Do
6     index = (21 * i) % 55;
7     ma[index]=tmp2;
8
9     tmp2 = (tmp1 - tmp2) % MBIG;
10    tmp1 = ma[index];
11   end
12
13   For j in 1..4 Do
14    For k in 1..55 Do
15          ma[k] = (ma[k] - ma[1+((k+30) % 55)]) % MBIG;
16    end
17   end
18
19   inext = 0;
20   inextp = 21;
```

Figure 28 System.Random initialization algorithm

Stepping the generator: stepping the generator is detailed in the following pseudo code.

```
1    inext = (inext % 56) + 1;
2    inextp = (inextp % 56) + 1;
3
4    outputNum = (ma[inext] - ma[inextp]) % MBIG;
5
6    ma[inext] = outputNum;
```

Figure 29 Stepping the System.Random generator

Choice of m: it's not documented why the implementers decided to use a different MBIG then Knuth's or the one in numerical recipes. It could be that this was chosen to be the biggest positive number for 32 bits integers in order to support returning more bits to the user in a single cycle of the generator or due to the point showed in [62] and [92] regarding skews in the bit distribution tests of bits 22-29.

The selection for *k* is **not understandable** and it is a flaw in the algorithm because it causes the polynomial of the generator to be reducible, as can be seen in the "short cycles attack" described in 7.2.3.1.

Period: due to the fact that the polynomial of the generator is reducible, the period is input dependent and does not achieve the maximal possible period for all inputs.

State: The state size is 55 values of 31 bits values that are stored in an array, so the total size of the state is 55*31=**1705** bits.

Output: The value returned to the caller at step n is a 31 bit value, which is simply X_{n-55}.

Entropy use: this implementation **doesn't** add any entropy to the generator.

Seed: the only option to set the seed is during first instantiation of the Random object. The seed is an integer value of 32 bits, which is expended to the whole state array in the initialization phase.

Default seed implementation: the default seed takes the current tick-count, by invoking *Environment.TickCount*. This value is 32 bits integer and represents the amount of **milliseconds** that passed since the computer was **restarted**, which means the computer's up-time.

7.2.3 Properties Analysis

7.2.3.1 Pseudo-randomness

Subtractive random generators general note: Knuth states in [1], page 28, that one should think carefully before using this PRNG implementation since it doesn't rely on a strong proven theoretical background as other generators and that a lot is known about LCG's properties whereas not much is known about the subtractive method. He also states that this method has passed all the statistical tests and has a good long cycle when used with correct parameters.

Small cycles in lsb bit due to bug in the implementation: there is a bug in the implementation in C# of Knuth`s algorithm. The bug causes the construction to not behave like Knuth pointed out, thus it doesn't necessarily guarantee the long full cycle. We will show a theoretical analysis of short cycles' existence in the lsb bit. We will also show concrete examples of inputs to a slightly modified version of the generator that indeed have a very short cycle in this lsb.

The use of **21** instead of **31** for the *k* parameter (the distance between the two pointers) causes the generator to be with a generating polynomial of $G(x) = \boldsymbol{x^{55} + x^{21} + 1}$ instead of Knuth`s polynomial of $\boldsymbol{x^{55} + x^{24} + 1}$. This polynomial **doesn't** satisfy the requirements of being primitive. **Thus we don't get the guarantee of a full cycle**. Furthermore, as we'll see this polynomial **isn't** even irreducible.

By running the code in http://code.google.com/p/rabinfingerprint/source/browse/trunk/src/org/bdwyer/galoisfield/Polynomial.java?r=5 we found the irreducible polynomials that **factor the polynomial *G(x)***. These polynomials are:

1. $F_1(x) = H(x) = x^9 + x^6 + x^4 + x^3 + x^2 + x + 1$
2. $F_2(x) = L(x) = x^{19} + x^{15} + x^{14} + x^{13} + x^{12} + x^8 + x^5 + x^3 + x^2 + x + 1$
3. $F_3(x) = K(x) = x^{27} + x^{24} + x^{23} + x^{21} + x^{20} + x^{19} + x^{16} + x^{14} + x^{12} + x^{10} + x^7 + x^6 + x^4 + x^2 + 1$

In the next discussion we will see how this **reducible** polynomial G(x) **affects the lsb bit under the simplification that MBIG is even (and not odd as in the implementation)**. (This is done because adding MBIG which is odd, as opposed to the one that was used in *numerical recipes in C* and in Knuth, changes the lsb and makes the analysis more complicated). It seems like a relevant assumption to the analysis that emphasizes how fragile the current implementation is.

We **denote *S*** as the state consisting of only the lsb bit of each state element, thus $|S|=55$ bits. We verified that polynomials *H(x)*, *L(x)* and *K(x)* are **primitive**, thus we know that each polynomial induces a maximal cycle of $2^{deg} - 1$, where *deg* is the degree of the polynomial. We denote each of these cycles as *cycle(H)* as the cycle of *H, cycle(L)* as the cycle of *L* etc. We can consider each state variable, S_i, as a triplet of the projections of S_i over *G*`s polynomial factors:

$$S_i \bmod G(x) = (S_i^H, S_i^L, S_i^K) = (S_i \bmod (H(x)), S_i \bmod (L(x)), S_i \bmod (K(x)))$$

Reminding ourselves that **stepping** an LFSR is in fact performing *X * S(x) mod G(x)* gets us to the equivalent representation of:

$$X*S_i \bmod G(x) = X*(S_i^H, S_i^L, S_i^K) = (X*S_i \bmod (H(x)), X*S_i \bmod (L(x)), X*S_i \bmod (K(x)))$$

So **after k steps** from the initial states we get:

$S_k \bmod G(x) = X^k * S_0 \bmod G(x) = X^k * (S_0^H, S_0^L, S_0^K) =$

$$(X^k * S_0 \bmod (H(x)), X^k * S_0 \bmod (L(x)), X^k * S_0 \bmod (K(x)))$$

So the **cycle length k** is the minimal number for which $X^k mod F_j(x) = 1$ for each j=1...3. This is exactly the **LCM of the orders of x in the field induced by F_j,** where $S_0 mod F_j \; ! = 0$.

Below are the cycle lengths histogram, where we divide the seeds to buckets according to being divisible by G(x)'s factors. The cycle length of each bucket is the LCM of the cycle lengths corresponding to the polynomials that **don't divide the seeds**.

Scenario	S_i^H	S_i^L	S_i^K	# seeds	Cycle Length
1	0	0	*	$2^{55-9-19} = 2^{27}$	$cycle(K) = 2^{27} - 1$
2	0	*	0	$2^{55-9-27} = 2^{36}$	$cycle(L) = 2^{19} - 1$
3	*	0	0	$2^{55-19-27} = 2^{46}$	$cycle(H) = 2^9 - 1$
4	0	*	*	$\cong 2^{55-9}$	lcm(cycle(K), cycle(L)) = $(2^{27} - 1)*(2^{19} - 1) \cong 2^{46}$
5	*	*	0	$\cong 2^{55-27}$	lcm(cycle(L), cycle(H)) = $(2^{19} - 1)*(2^9 - 1) \cong 2^{28}$
6	*	0	*	$\cong 2^{55-19}$	lcm(cycle(K), cycle(H)) = cycle(K)= $2^{27} - 1$
7	*	*	*	$2^{55} - (all\ of\ the\ above)$	lcm(cycle(K), cycle(L), cycle(H)) $\cong 2^{46}$

Figure 30 Cycle Length Histogram

Finding concrete inputs: we managed to find concrete inputs that would get the generator (having even MBIG) to have these shorter cycles.

1. If we ignore the initialization process that was described in 0 and directly set the initial state lsb bits to be { 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0 } (msb first) we get a $\mathbf{2^9 - 1}$ cycle. This input was found by taking K(X)*L(X). We note that brute force won't work since the probability of getting a state with such short cycle is $(\frac{1}{2})^{27+19} = (\frac{1}{2})^{46}$.
2. We also find a seed, **before** the initialization process, that its expansion after the initialization phase corresponds to a polynomial which is a multiply of H(X)*L(X) so it has a $\mathbf{2^{27} - 1}$ cycle. This input is $(47368)_{10}$ and it was found by performing a brute-force over the possibilities for the seed, which was doable because the probability of such input is $(\frac{1}{2})^{9+19} = (\frac{1}{2})^{28}$.

We continue to outline more attacks regarding the general implementation of the algorithm, and not regarding the reduced cycle length.

Assuming we know the implementation is based on *System.Random*: this will give us the information of the algorithm used for the generator and its parameters.

Brute Force: a brute force attack of the state would require searching a space of $\mathbf{2^{55*31} = 2^{1705}}$ possibilities, which is a very big search space. Following this the next attack assumes we can get outputs from the generator.

Known cipher-text attack: consider we have the ability to observe integer outputs from this generator. For instance, consider an application that uses this generator to produce an ID that is visible to the user of the application. Recall that the generator simply outputs the number which will now be in index *inext*. This leads us to the fact that **after getting 55 consecutive outputs** from the generator we have the entire state in our hands. From this point on, we are in-sync with the generator. We note that the assumption of getting consecutive outputs isn't a very restrictive assumption.

Not getting consecutive outputs: in the attacks above we required getting 55 **consecutive** outputs. We can further relax this assumption (of consecutive outputs) since we know the parameters of the generator; we only require knowing the **position of the indices** during the output. If we were to know this, we can construct linear equations that would end up revealing the entire state.

Assuming we don't know the implementation is based on *System.Random.* If we allow our attacker to **gather 55 consecutives outputs** from the generator, then we can just try to run the algorithm and see whether we are synced with the generator. It is easy to see that we can achieve a distinguisher with only **3 outputs**: X_n, X_{n-34}, X_{n-55}. After getting these outputs we can verify that these 3 outputs fulfill the recurrence formula. The amount of false positives here is very low (~2^{-32}), so only these outputs suffice.

7.2.3.2 Backward Security

None (not entropy based).

7.2.3.3 Forward Security

None – if we have the state in our hands, we can reconstruct the subtraction equations in order to get to a previous state.

7.2.3.4 Seed Weakness

Using time variations as the seed isn't secure enough. Here the seed is the system's "up-time". We will show the entropy we get in the default seed isn't enough.

Brute force: as the seed is only represented by a 32 bit integer, the attacker needs to break a search space of $\mathbf{2^{32}}$, which using hardware found today is quite feasible.

How many entropy bits do we have in the default seed? The same techniques that we used in 6.3.3.4 in order to estimate the entropy are applicable here. For the sake of completeness, we'll briefly repeat both techniques with different parameters values to adhere to this implementation:

(a) Application startup – knowing the exact application startup time can easily be achieved by nmap probes or other techniques, if the server responds to these queries. Using this technique we can pin-point the server`s uptime. Even if we can't use these tools, we can get the attack to an order of days, which a day has about $\mathbf{2^{26}}$ milliseconds. Moreover the attacker could try and force a restart, thus limiting the amount of entropy the seed has.

(b) New objects for each invocation – assuming the attacker can guess the exact time of the server up to **one minute of error** and with the assumption we can know the server uptime, as described in (a), we can result in having to break only: $\log_2 (1_{min} * 60_{sec} * 1000_{ms}) = 15.87 \sim= $ **16 bits**.

We note that the very limited search space of the default seed makes the brute-force approach the easiest to mount, as it requires no a-priori information regarding the server current uptime.

7.3 System.Security.Cryptography.RandomNumberGenerator

http://msdn.microsoft.com/en-us/library/system.security.cryptography.randomnumbergenerator.aspx

7.3.1 Design Space

Much like in Java's JCS, .NET has its own cryptographic services implementation; the framework is implemented in the *System.Security.Cryptography* namespace.

System.Security.Cryptography.RandomNumberGenerator is an abstract class that is the base class for every RandomNumberGenerator implementation. Much like Java's JCS framework, here this framework allows building custom RandomNumberGenerator implementations.

The API is rather straightforward, and provides the following methods.

```
1    // Creation methods
2    protected RandomNumberGenerator();
3    public static RandomNumberGenerator Create();
4    public static RandomNumberGenerator Create(string rngName);
5
6    // Generator access API
7    public abstract void GetBytes(byte[] data);
8    public abstract void GetNonZeroBytes(byte[] data);
```

Figure 31 RandomNumberGenerator API

Drop-in replacement for *System.Random*: unlike in Java, where the stronger variant of the random generator, *SecureRandom*, still implements the same API as its weaker counterpart, *java.util.Random*, here the API for the RandomNumberGenerator implementer is different than the one of *System.Random* and the CLI doesn't provide any adapters between the methods. This forces the developer to add her own adapter methods in order for the implementation to be a drop-in replacement for *System.Random* implementation. Although this sounds like a minor issue we note that this is in fact a problem that troubles developers as shown in [93] and we believe can even get developers to use the weaker *System.Random* due to the richer and more convenient API provided there.

Resolving: much like in Java, there are several ways to resolve a generator provider. This is documented in [94]. The main ways are: (a) create the implementer by using the **new** operator, (b) use the *Create* method in the *RandomNumberGenerator* class or (c) use the *Create* method with the explicit name of the provider. A configuration exists in order to control providers by having the algorithm names associated with the algorithm provider implementation class.

GetNonZeroBytes: another interesting observation in the API is the existence of a **separate** method is order to get random bytes that do not contain any zeroes. According to [93] the only reason for having this method is to support PKCS #1 padding for RSA encryption. We note that the decision to have this method in the class design is **risky**: non careful user could use it instead of the regular one, which will result in a reduced randomness. The reduced randomness exists since we lose a **basic property** of a random stream, which says that each bit should have an equal probability (=1/2) of being 0 or 1. In fact, here, after seeing 14 bits of 0, we can conclude in a 100% probability that the next bit will be 1. Since this method isn't intended to be used, we omit this evaluation from the properties analysis of this generator.

The provider shipped with the CLR is Microsoft's default (and only) provider implementation, ***System.Security.Cryptography.RNGCryptoServiceProvider***. This implementation falls quickly to unmanaged code to generate the random values.

Implementation of the default provider: by having the source code from the SSCLI, we managed to get the implementation used in this provider. The implementation delegates the main method, *GetBytes*, to the method ***CryptGenRandom***. Since the code path is not straightforward, we

will continue to describe the code path that leads us to this conclusion. *GetBytes* is delegated to *Win32Native.Random* method, which exists in *win32native.cs* source file. This method is implemented in the *win32* native platform DLL with an entry point of *PAL_RANDOM*. This method has the following signature:

```
1    bool Random(bool bStrong, byte[] buffer, int length);
```

The parameters in this method are an output buffer, length and Boolean flag indicating whether it should use the strong variant. The provider invokes this method by always passing a *true* value in the *bStrong* parameter and the user supplied output *data* buffer as the buffer. The implementation of the method exists in the native C file, *win32pal.c*. The implementation then tries to acquire a context to the Crypto services, by using the *CryptAcquireContext* function. After successfully acquiring the context the implementation invokes the ***CryptGenRandom*** function with the given buffer to fill.

Acquiring *CryptAPI*: the implementation always tries to acquire a handle to the CryptAPI which is a bit redundant since in most cases we only wish to use the PRNG, without the loading of the entire API implementation. The implementers could have implemented it differently, as pointed in [95]. This is not a problem per-se, only a redundant memory overhead for the invoking application. This is a minor overhead, since the implementation does hold the handle only once as can be seen in the *win32pal.c* source file.

For complete applicable source code as extracted from the SSCLI, please refer to 11.2.3.

7.3.2 Under the Hood

Windows platform/version dependence: we concluded in the design section that the generator uses the Windows Random Number Generator, or *WRNG*. However there was a major change of the implementation of the WRNG during Windows releases. From code de-compilation performed on the DLLs *advapi32.dll* and *rsaenh.dll* on a Windows 7 32 bit platforms (using the IDA-Pro disassembler); it shows that the implementation is based on AES. The function is called *_AesCtrWithFipsChecks* and is implemented in the *rsaenh.dll*. This is also mentioned in the *CryptGenRandom* MSDN entry, which points out that from Windows Vista SP1 the implementation uses an implementation of the AES counter-mode based PRNG specified **in NIST Special Publication 800-90** (see 2.4.10.1 for some details)**.**

In Windows machines **earlier** than Windows Vista SP1, the WRNG was implemented using the algorithm specified in **FIPS-186-2 appendix 3.1** [29] construction, with the use of SHA-1 as the G function.

The generator used is the Windows Random Number Generator: at the time of writing this paper we are aware of only one detailed documentation and analysis of the *WRNG* that was published by Leo and Gutterman [53]. Unfortunately, their analysis was conducted on a Windows 2000 machine, which only has the old, FIPS-186-2, generator. The *WRNG* analysis is summarized in section 3.8.

Setting an external seed: We observe that the documentation of .NET standard *RandomNumberGenerator* API doesn't specifically state how to accomplish this. The documentation of *CryptGenRandom* [96] states that the buffer passed is actually an IN/OUT buffer and the data it contains is used as an auxiliary seed. This is quite confusing, as most developers would simply look at the immediate *RandomNumberGenerator* API to check how to give their own seed and not in the *CryptGenRandom* API. This is an error in the documentation as shown in [53]. Leo further states that according to the algorithm analysis, the *Buffer* passed to the function is actually treated as if it is empty (since we concatenate the value *T* to it) and there isn't any consideration of this input

buffer in the generator's output. Following this we conclude **that there isn't any option to use an external seed to the generator**.

We note that allowing a user-defined input to a generator is a recommended functionality, as described in NIST 800-90, section 7.2, page 12: "This Recommendation strongly advises the insertion of a personalization string during DRBG[8] instantiation…"

Default seed: the seed is generated from system entropy sources, in the rekey process of each of the RC4 generators, in the **get_next_20_rc4_bytes** method.

State: as mentioned in 3.8, the state of the generator is dictated by the two variables, *R* and *State* and the states of the 8 *RC4* generators. This gets us to a state size of **40+8*256=2,088** bytes.

Period: similar to the *WRNG*, the algorithm used relies on various entropy sources and utilizes an invocation of a SHA1 variant. Discussing the period for this implementation is irrelevant.

7.3.3 Properties Analysis

7.3.3.1 Pseudo-randomness

As our paper tries to only discuss cipher based attacks, in which we don't assume the attacker has a way of accessing the machine or accessing the state as a whole, we do not know of any known cipher attack on the *WRNG*. We can only state that obvious **brute force** attack over the state would require an effort of $2^{16,704} = \mathbf{2^{40*8+256*8*8}}$ due to the size of the internal state.

The rest of the analysis is shown in 3.8; there we show the attacks that the authors in [53] found in the *WRNG* implementation.

7.3.3.2 Backward Security

See 3.8 for details.

7.3.3.3 Forward Security

See 3.8 for details.

[8] deterministic random bit generator.

8 PHP

8.1 Introduction

The following analysis is based on version 5.2.3 of the PHP interpreter and runtime libraries [97] dated 31/5/2007. The PHP engine is code-named *Zend* and is implemented as a mixture of C and PHP code. All of the generators are implemented in C.

Popularity of PHP: PHP is the most popular programming language for server side scripting. According to [98], as of November 2006 there were more than 19 million websites using PHP. Furthermore, according to the TIOBE index [3] PHP is the **4th** most popular programming language.

Windows vs. Linux: since the common scenario for deploying a PHP based application is on Linux machines, we'll consider this as the platform of this analysis.

Functions available: there are two official library PRNG functions called ***rand()*** and ***mt_rand()***. There is also another function, ***lcg_value()***, which is used internally by the PHP engine. This function is also available to the user, however it's not referenced in the documentation from the *rand()* or *mt_rand()* documentation.

Documentation: unlike previous languages covered, PHP's documentation doesn't state the **exact** implementation of its default PRNG *rand()*. Furthermore, the PHP documentation of the weaker functions doesn't state the concerns and warnings regarding using some of the generators for cryptographic purposes.

Operating System PRNGs support: unlike other languages we've covered, in the standard PHP implementation there isn't a PRNG that allows the user to use an Operating System based PRNG. In the Zend engine implementation we found usages of an Operating System PRNG in Zend's engine memory heap protection. The implementation uses CryptGenRandom on Windows machines and /dev/urandom on supported platforms. The source code can be viewed in the file *zend_alloc.c.* However these usages were sporadic and weren't externalized as a PHP API function for ease of use for developers. We note that, naturally, the user has a way to invoke these PRNGs, either as reading from a file (under Linux) or invoking the CryptGenRandom Win32 API method using a PHP Zend C plugin.

Scope of analysis: This analysis will not cover several other runtime libraries which are shipped with PHP, e.g., *sqlite* (which uses an RC4 based PRNG) and *openssl*. Moreover, there is a special library for manipulating arbitrary length numbers, called GMP and is based on the GNU GMP (Gnu Multiple Precision Arithmetic library) (http://gmplib.org/). This is also not covered in the analysis. Like many other language engines and runtime implementations, the PHP implementation has different implementation for thread-safe functions (reentrant versions). The thread safe functions in the Zend implementation are enclosed with a C *define* named *ZTS* (Zend Thread Safe). Since most deployment of PHP use the Apache [99] webserver in pre-fork mode, which utilizes processes as isolation between requests, we won't cover these variants in this analysis. We will also leave covering the flavor *mt_rand()* for future work, in an effort to keep this work concise

Difficulty of analysis: the Zend engine is a complicated virtual machine implementation for PHP. We used the documentation in [100] as a reference reading for understanding the implementation alongside the source code available to download. Functions defined in Zend do not accept parameters as normal functions in C. The parameters are manipulated on a special **PHP specific stack**; this caused the analysis to be far from straight forward.

8.2 lcg_value() PRNG

http://php.net/manual/en/function.lcg-value.php

8.2.1 Design Space

The function *lcg_value()* is implemented in the *lcg.c* source file and declared in the *php_lcg.h* header file. The actual implementation is in *php_combined_lcg* function.

This generator is used in numerous places within the Zend engine code; it is also used in the default seed generation for other API generators, as can be seen in the next sections. As can be seen in 10 it is also an important building block in PHP's session generation algorithm.

API: The function doesn't expect any parameters and returns a random value in the range of (0, 1).

8.2.2 Under the Hood

The implementation is based on the generator implementation introduced by L`Ecuyer in [20] that was covered in section 2.4.3. The implementation is identical to the one showed in [21] chapter 16.1. The documentation of the implementation doesn't refer to any of the above as references.

The theoretical PRNG behind it is a Combined LCG: it combines two MLCGs by subtracting both states. The formula follows:

MLCG1:
$s_{n+1}^1 = (s_n^1 * a_1)\ mod\ m_1 = (s_n^1 * 40014)\ mod\ (2^{31} - 85)$

MLCG2:
$s_{n+1}^2 = (s_n^2 * a_2)\ mod\ m_2 = (s_n^2 * 40692)\ mod\ (2^{31} - 249)$

Combined MLCGs:
$z_{n+1} = (s_{n+1}^1 - s_{n+1}^2)\ mod\ (m_1 - 1) = (s_{n+1}^1 - s_{n+1}^2)\ mod\ (2^{31} - 84)$

The source code of the generator doesn't use the straight forward calculation as written above in order to be portable between platforms.

Parameters: the parameters used for MLCG1 and MLCG2 are chosen in a way that guarantees the maximal period of *Z*.

Output calculation: the output is calculated from Z by multiplying Z_{n+1} by the inverse of m_1. The output is calculated as follows:

```
1    output = (z_{n+1} < 0) ? (z_{n+1} + m_1 - 1) : z_{n+1};
2    output = output/m_1 = output * 4.656613e - 10;
```

Figure 32 Output calculation of Z

Period: L`eculyer, in [20] chapter 5, shows that the period of this generator is

$$\frac{(2^{31}-85-1)*(2^{31}-249-1)}{2} \cong \mathbf{2^{61}} \cong 2.3 * 10^{18}$$

State: the state is the state of both MLCGs; each of the MLCGs holds a state variable of roughly 31 bits. This leads to a combined state size of **62 bits**.

Seed: there **isn't** an option to externally set the seed. The implementation makes sure the generator is seeded by setting the seed using the default seed implementation.

Default seed implementation: the default seeding algorithms seeds both MLCGs. The seeding process uses time and process id inputs as the seed. The pseudo code for the initialization follows.

```
    // Initializing MLCG1
1   if (gettimeofday is available) then
2
3     c = gettimeofday();[9]

4     s_0^1 = c.seconds XOR NOT(c.micro-seconds);
      // NOT(c.usec) is the bitwise NOT of microseconds
5
6   else
7     s_0^1= 1;
8   end
9
10  // Initializing MLCG2
11  s_0^2 = getpid();[10]
```

Figure 33 MCGs initialization algorithm

Size of the default seed: we continue to observe the size of each of the MLCGs default seed size.

gettimeofday function: *gettimeofday* puts time data in its associated struct from the Epoch time (1/1/1970 00:00). The seconds are saved in the *tv_sec* field and the micro-seconds are saved in the *tv_usec* field and represent the amount of micro-seconds modulo 1 second, meaning the amount of microseconds within the *tv_sec* value. Both fields use an integer to hold the values. The *seconds'* field can be up to 32 bits and the *microseconds'* field can be up to 20 bits[11] (on a 32 bit machine); this gets the s_0^1 value to be **32 bits** in size.

Maximum number of Process IDs on a Linux platform: by default Linux wraps around the process identifiers when they exceed **32768** (2^{15}). An administrator can increase this value up to 2^{22} on 32 bits machines; however it's not common that one would need to configure maximum number of processes to be this high. We also note that the limit per user is even lower than this value, and defaults to 1024 (as can be seen by running the command *ulimit –n*). Following the last note regarding the maximum number of process IDs we can conclude that the size of s_0^2 is **15 bits**.

Scoping: we note that the seed of the *lcg_value* generator is globally defined and instantiated per PHP script. This means that revealing the seed within a specific script is applicable for every invocation of *lcg_value* in that script scope. The seed is initialized with the **first** invocation of *lcg_value.*

Entropy use: this implementation **doesn't** add any entropy to the generator.

8.2.3 Properties Analysis

8.2.3.1 Pseudo-randomness

Assuming we know the implementation is based on *lcg_value()*: this gives us the information regarding each MLCG and the fact that this algorithm is used.

Our goal is getting to the states of both generators, MLCG1 and MLCG2. Since both generators are in fact MLCG, they don't poses the forward/backward security properties; thus after knowing both states we can easily traverse forward or backwards in both generators' streams to our liking.

Brute Force: a brute force attack requires going over the entire space of the state: 2^{62}.

[9] On a Windows platform we don't have native support for gettimeofday function. The PHP engine implementation has its own implementation for this for Win32. Can be seen in time.c and time.h

[10] In case we are in the multi-threaded implementation, meaning ZTS is defined, this would be the thread-id.

[11] The precision of this timer depends on the platform used. According to http://stackoverflow.com/questions/88/is-gettimeofday-guaranteed-to-be-of-microsecond-resolution on Intel processors the micro-second precision is guaranteed, but on some other platforms it can be as low as 10 micro-seconds.

We follow with a more efficient attack that uses several outputs of the subtraction, z. We note that we **ignore** the **final step of the output** that gets z to be a fraction between (0, 1) as it is easily reversible. We will now show how we can effectively reverse the subtraction and get the states of both generators.

Known cipher text attack: we assume that we can get at least **3 consecutive outputs** from the generator. Let's write the equations for 2 consecutive outputs of the generator:

$$(*)\ z_n = (s_n^1 - s_n^2)\ mod\ (m_1 - 1)$$

$$z_{n+1} = (s_{n+1}^1 - s_{n+1}^2) mod\ (m_1 - 1) \xRightarrow{replacing\ with\ MLCG\ equations}$$

$$(**)\ z_{n+1} = ((s_n^1 * a_1\ mod\ m_1) - (s_n^2 * a_2\ mod\ m_2))\ mod\ (m_1 - 1)$$

z modulo: We denote $z_n^{internal}$ and $z_{n+1}^{internal}$ as the values **before** performing the $(m_1 - 1)$ modulo. Given z_n, z_{n+1}, we have four possibilities for $z_n^{internal}$ and $z_{n+1}^{internal}$:

(1) $[z_n, z_{n+1}]$
(2) $[z_n - m_1 - 1, z_{n+1}]$
(3) $[z_n, z_{n+1} - m_1 - 1]$
(4) $[z_n - m_1 - 1, z_{n+1} - m_1 - 1]$

Our attack will go over all those 4 possibilities, so from now on, we'll omit the *mod (m_1-1)*.

We note that if both MLCGs were to use the same modulus ($m = m_1 = m_2$), then solving (*) and (**) to find $s_n^1\ and\ s_n^2$ from z_n, z_{n+1} would be simply a matter of solving two linear tions $mod\ m$.

We'll now show how to solve (*) and (**) to find $s_n^1\ and\ s_n^2$ from $z_n^{internal}\ and\ z_{n+1}^{internal}$ in the real situation of having different m1 and m2.

Equivalent representation the equations above can be written as:
(1) $z_n^{internal} = s_n^1 - s_n^2$
(2) $z_{n+1}^{internal} = (s_n^1 * a_1 - k_1 * m_1) - (s_n^2 * a_2 - k_2 * m_2)$

We also get the following requirements:
(3) $0 \le s_n^1 \le m_1$
(4) $0 \le s_n^2 \le m_2$
(5) $0 \le s_n^1 * a_1 - k_1 * m_1 \le a_1$
(6) $0 \le s_n^2 * a_2 - k_2 * m_2 \le a_2$
(7) $s_n^1, s_n^2 \in N$

By extrapolating s_n^2 from (1) we get $s_n^2 = s_n^1 - z_n^{internal}$, by substituting this in equation (2) we get:

$$z_{n+1}^{internal} = s_n^1 * a_1 - k_1 * m_1 - \left(s_n^1 - z_n^{internal}\right) * a_2 + k_2 * m_2$$
$$z_{n+1}^{internal} = s_n^1 * a_1 - k_1 * m_1 - s_n^1 * a_2 + z_n^{internal} * a_2 + k_2 * m_2$$
$$z_{n+1}^{internal} - z_n^{internal} * a_2 - k_2 * m_2 + k_1 * m_1 = s_n^1(a_1 - a_2)$$

Finally, we extrapolate s_n^1 and get:

$$s_n^1 = \frac{z_{n+1}^{internal} - z_n^{internal} * a_2 - k_2 * m_2 + k_1 * m_1}{a_1 - a_2} \qquad (*)$$

$$s_n^2 = s_n^1 - z_n^{internal} \qquad (**)$$

A simple solution of (*) and (**): first we'll introduce a naïve solution and we'll follow with more efficient one. In order to solve the equations we'll run over all possibilities of $\boldsymbol{k_1 * k_2 * 4}$ (*4 due to the four possibilities $z_1^{internal}, z_2^{internal}$ as mentioned above), solve (1) and (2) and make sure all requirements and equalities outlined in (3)-(7) are met. This gets us to the conclusion that after getting two **consecutives** outputs we can effectively reveal the entire state of both generator (i.e. s_n^1 and s_n^2) in an effort of $\boldsymbol{k_1 * k_2 * 4}$ operations. From (5) and (6) we know that $k_1 \leq a_1$ *and* $k_2 \leq a_2$ so we need to do $\boldsymbol{a_1 * a_2 * 4}$ operations, which mean an effort of $\boldsymbol{2^{32} < 40014 * 40692 * 4 < 2^{33}}$. This is much better than the brute-force approach that would require an effort of 2^{62}.

Validating our solution: we are trying to find a state of **62 bits** and we have an output of 62 bits, so information theory suggests that we will have only a few possible solutions. So one more output will be enough in order to validate which solution is the correct one; thus the use of the third output.

A more efficient version of solving (*) and (**): we observe that $s_n^1 < m_1$; this means that we can safely perform a modulo of m_1 operation on (*) without losing information on s_n^1. This gets us to the following revised (*) equation:

$$s_n^1 = (z_{n+1}^{internal} - z_n^{internal} a_2 - k_2 m_2) * (a_1 - a_2)^{-1} \, mod \, m_1$$

As we can see taking modulo of m_1 removed the dependence of k_1 and we only need to iterate over all the possibilities for k_2. This gets us to needing an effort of only $\boldsymbol{k_2 * 4 = 40692 * 4 = 162,768 \cong 2^{17}}$. Note that now we need to calculate $(a_1 - a_2)^{-1} \, mod \, m_1$, which can be done efficiently. The remainder of the solution procedure is the same as before (i.e. verifying (3)-(7) for each s_n^1, s_n^2 we get when iterating over k_2).

Assuming we don't know that the above generator is used. If we were to allow ourselves to get **3 consecutive** outputs, then we can try and solve the equations as outlined above as a distinguisher for this generator.

8.2.3.2 Backward Security

None (not entropy based).

8.2.3.3 Forward Security

Since both generators are MCGs, they don't have the forward security property.

8.2.3.4 Default Seed Weakness

For this analysis we'll assume the *gettimeofday* function is available. Otherwise, the seed for the first generator is simply a constant.

Let's examine how much entropy we actually have in both seeds; we'll use s_0^1 and s_0^2 to denote the seeds of both generators respectively.

As in other analysis of time variants we carried throughout this paper, we can assume that the seed is generated either at the startup of the application or during the first use of *lcg_value*.

s_0^2 **(process-id):** as mentioned, the actual entropy of this seed is far from the required 31 bits. It is merely 15 bits of entropy. However this can be reduced even more with the following observations.

Linux Process ID Allocation Algorithm: there is a new process ID allocation algorithm in Linux as can be seen in [101,102,103]. The old algorithm used to increment the process-id until reaching the */proc/sys/kernel/pid_max* value and then wrapping around, finding empty slots to fill. In an empirical test, we verified that the new algorithm still allocates processes in a sequential manner. We verified this in the kernel source code version 2.6.33 dated 24/2/2010; the new algorithm is used and implemented in *pid.c.* There are many known processes that are allocated during system startup. This means that the actual entropy of this value is even lower. By observing, for example, an actual Linux server deployment that utilizes Apache [99] as its webserver and running on Redhat Linux [104] we saw that almost all processes up to roughly PID=5000 are system processes. This can serve as the attacker lower process-id bound ps_{min}. This gets us to effective entropy of **14 bits.**

$s1_0$ (output of gettimeofday): as in other analyses carried out throughout this paper, we can assume that the generator was seeded close to the application startup, and thus server startup. If we assume this we can pin point the server startup to a specific day in a year, then the entropy will be about 16 bits. Nevertheless, we still have the micro-seconds within this day, which means another 20 bits. Because we xor those two numbers, and because the variable part of both of them is in the lsb, the entropy of the xor output has **only 20 bits**. Note that if another operation was performed instead of xor one could have achieved the maximum entropy of 32 bits.

In 10 we show a concrete attack on the session ID generation in PHP that utilizes our attack on the pseudo-randomness of this generator.

8.3 rand() PRNG

http://php.net/manual/en/function.rand.php

8.3.1 Design Space

The implementation of the *rand/srand* function resides in *rand.c* source file and declared in *php_rand.h* header file. The implementation of the API functions is located in *php_rand()* and *php_srand* functions.

API: the ***rand()*** function accepts two **optional** parameters: *min* and *max*. The function returns a random number from the range [min, max], whereas the default values are *min=0* and *max=RAND_MAX*. Like in the C implementation there is a complementary seeding function called ***srand***. The caller doesn't have to call the seeding function and the runtime makes sure the generator is initialized upon calling *rand*. For further details regarding the seed, refer to 8.3.3.2 section.

8.3.2 Under the Hood

There isn't a native implementation for the generator, and the implementation resolves to an appropriate C implementation. The resolving order follows.

Resolving: the implementation **first** checks if the platform supports the ***random*** functions family that were covered in 5.4. If not found, it searches for the ***lrand48*** flavor that were covered in 5.5. If not found, it falls back to the default C ***rand()*** implementation that was covered in 5.3.2.[12]

RAND_MAX: the *RAND_MAX* constant, the maximum number that the generator can return to the user, also depends on the flavor used: in C *rand* and in the reentrant flavor it is 2^{15}, in the other variants it is 2147483647 ($2^{31} - 1$).

Seed: there is an option to externally set the seed, using the *srand* PHP API function. The function uses the appropriate seeding function according to the resolving specified above.

Scoping: much like in the *lcg_value* generator, the seed for the *rand* function is globally saved per PHP script in its own variable.

Default seed implementation: the implementation implements a default seed, regardless of the flavor used. The algorithm for generating the seed is defined in a macro called *GENERATE_SEED* that is defined in php_rand.h file. The algorithm performs the following steps:

1. **Time** ($T_{seconds}$) - takes the current time using the *time()* function. The *time()* function returns the amount of time **in seconds** that had elapsed since the Epoch.
2. **Process ID** (pid) **–** takes the current process ID using the get*pid()* function.
3. **lcg_value/php_combined_lcg** (lcg_value) – takes a single output from the *lcg_value* generator that was covered in 0.

The output of these steps is a multiplication to yield 32 bits in the following way:

```
1    S0 = Tseconds * pid * 1,000,000 * lcg_value;
```

Figure 34 PHP rand() default seed algorithm

The size of this default seed is **32 bits**. If the random() or the rand() flavor is used, this value is cast to an *unsigned int*.

8.3.3 Properties Analysis

8.3.3.1 Pseudo-randomness

All properties analysis of the generator depends on the C flavor used. The reader is encouraged to go to the appropriate sections in order to read the analysis.

[12] In case ZTS is used, the implementation uses the *rand_r* implementation.

8.3.3.2 Default Seed Analysis

We examine how many entropy bits we have in the **default seed.** As we showed in 8.2.2, the PID has an entropy value of **14** bits. Assuming we can pin point $T_{seconds}$ *exactly* (like always – it's either system startup or current time) this has no entropy. We are left with the output of *lcg_value*, which can have an output of maximum **31 bits**.

lcg_value default seed: remembering that the default seed of the *lcg_value* is also comprised of the process-id and time variations, we can further try and limit the entropy. Since this is the same process ID, this doesn't add entropy. However, the default seed of lcg_value also utilizes micro-seconds precision using the *gettimeofday* function. Assuming we can know the system startup to a precision of more than a second is too hard of an assumption. This leaves us with the entropy of the micro-seconds field of **20 bits.**

Total: adding the 20 bits to the 14 bits of the PID yields 34 bits of entropy, which is reduced to **32 bits** due to the use of a 32 bits data type.

We note that if we were to allow our attacker to have stronger abilities, this can easily be reduced. As an example: if the attacker has access to the processes that run on the machine. This can give the attacker the knowledge of the PID.

The use of *lcg_value* in the default seed generation could yield some knowledge regarding the *lcg_value* generator. Considering a complex script (or a chaining of invocations between scripts) that utilizes both *rand* and the *lcg_rand* generator. Since the seed of the *lcg_rand* is the same between invocations in the scope of the same PHP script, getting an output from the *rand* generator can reveal data of the *lcg_rand* inner state. This is because this weak flavor has no forward security, so we can reverse the generator and get to the initial seed. From there we can try and get to the *lcg_value* output. This can potentially aid us in a different attack on *lcg_value*; e.g., by using this value to invalidate a guess.

9 Summary and Conclusions

In this work we presented a detailed analysis of the pseudo random number generators in the following programming languages: C, Java, C# and PHP.

In the introductory sections we provided the required theoretical common ground regarding PRNGs. Furthermore, we showed concrete examples of attacks on systems that were possible due to problems in the underlying PRNG. By showing these examples we tried to convey the importance of using a well thought off, studied generators (and seed generation algorithms) with proven security properties.

We presented a complete analysis of the PRNGs available in the above mentioned programming languages, complete with the analysis of their security properties. We demonstrate that the designers of each programming languages decided on different generators to be available to their users. The generators span from the LCG generator to Operating System based generators. We provided a strict evaluation of the strength of each implementation, including an analysis (if applicable) of their default seed implementation.

During the analysis we found a potential bug in the C# implementation of the subtractive random number generator. The bug causes the generator to stir away from sound mathematical theory and causes it to have shorter cycles than expected. We further show (in 10) a concrete attack of the session ID generation in PHP that was based on our analysis of the generator in PHP.

In summary, our impression is that programming languages SDK designers and developers should make more detailed (and correct) documentation available for developers. Where possible they should support flavors that rely on strong cryptographic building blocks (such as the recommendations in the NIST 800-90 standard) and well-studied ways to collect entropy for the seed. It's important to note that most usages of random numbers aren't only for security systems. Almost every developer, throughout her career would encounter the need to perform randomization of some sort. Our hope is that this work could be of guidance to her.

We summarize our findings in the following table.

Language	Flavor	Platform	PRNG	Entropy?	Period	State Size (bits)	Ext' Seed?	Default Seed	Security Properties			
									State	Default Seed	Forward	Backward
C (MSVCRT)	rand ANSI-C	Windows	LCG	No	2^{32}	31	Yes	Constant	2^{16}	N/A	No	No
C (MSVCRT)	rand_s	Windows	WRNG	Yes	N/A	16,704	No	System entropy sources (WRNG)	2^{16704}	N/A	2^{23} between rekeying	None between rekeying
C (glibc)	rand	*NIX	Invokes the generator in the BSD family random() function									
C (glibc)	BSD random G0	*NIX	LCG	No	2^{31}	31	Yes	Constant	O(1)	N/A	No	No
C (glibc)	BSD random G1	*NIX	AFG	No	$2^{31}(2^7-1)$	224	Yes	Constant	2^7	N/A	No	No
C (glibc)	BSD random G2	*NIX	AFG	No	$2^{31}(2^{15}-1)$	480	Yes	Constant	2^{15}	N/A	No	No
C (glibc)	BSD random G3	*NIX	AFG	No	$2^{31}(2^{31}-1)$	992	Yes	Constant	2^{31}	N/A	No	No
C (glibc)	BSD random G4	*NIX	AFG	No	$2^{31}(2^{63}-1)$	2,016	Yes	Constant	2^{63}	N/A	No	No
C(glibc)	SVID rand48	*NIX	LCG	No	2^{48}	48	Yes	None	2^{16}	N/A	No	No
Java	Random	Independent	LCG	No	2^{48}	48	Yes	Time	2^{48}	2^{30} or 2^{36}	No	No
Java	SecureRandom P1	Windows	WRNG	Same as in MSVCRT rand_s								
Java	SecureRandom P2	*NIX	LRNG + P3	Yes	N/A	1,184	Yes	System entropy sources (LRNG)	2^{1184}	Entropy	2^{64} or 2^{96}	2^{1184}
Java	SecureRandom P3	Independent	SHA1	No	2^{80}	160	Yes	System entropy or WRNG/LRNG	2^{160}	Entropy	2^{160}	No
Java	SecureRandom P4	PKCS-11	N/A – not covered in analysis									
C# (.NET)	Random	Windows	LFG	No	N/A	1,705	Yes	Time	O(1)	O(1) or 2^{16}	No	No
C# (.NET)	RandomNumberGenerator	Windows	WRNG	Same as in MSVCRT rand_s								
PHP	lcg_value()	*NIX	CMCG	No	2^{61}	62	No	Time and PID	2^{17}	2^{35}	No	No
PHP	rand()	*NIX	Falls back to the C implementation in the following order: BSD random, SVID rand48, ANSI-C rand							2^{32}	No	No

PHP	mt_rand()	Mersenne Twister	N/A – not covered in analysis

Figure 35 Analysis Summary Table

Notes regarding the summary table:

1. **Security Properties** – the security properties analysis is based on the analysis conducted in this paper. This means that we assume we can get one or more outputs from the generator.
2. **Platform** – the platform written per each variant is the platform that we used in the analysis. For instance, PHP is also available on Windows machines, however we conducted the analysis only on *NIX platforms.
3. **Period in C#'s System.Random generator** – due to the fact that we found a bug in the analysis (see section 7.2.3 for details) we couldn't properly analyze the period of this flavor.

10 Appendix A: Application Attack: Attack on PHP's Session ID Allocation

10.1 Introduction

As PHP is mostly used as server side scripting; much like .NET, it comes with its own session management module. Although this work's focus is to investigate the implementations of PRNGs in popular programming languages we also present here a novel and efficient attack on the session-id generation in PHP which uses the security analysis of *lcg_value* carried in the previous section.

10.2 Session ID Allocation Algorithm

The session ID allocation algorithm in PHP is implemented in the function *php_session_create_id,* in the *session.c* source file.

Allocating of a new session-id is performed during the procedure of handling new sessions. The allocation is performed using the following steps:

1. Get the **remote IP address** that this request originated from. The remote IP address is obtained by looking at the value of the HTTP header REMOTE_ADDR HTTP, which is available as standard execution environment information (http://php.net/manual/en/reserved.variables.server.php). In case the header does not exist, an empty string is used.
2. Get the **seconds** field of *gettimeofday*.
3. Get the **micro-seconds** field of *gettimeofday*.
4. Get a **single output** from the PRNG **generator** *php_combined_lcg* that was initialized with the default seed, which is the generator we covered in 0. The value is taken as a string after multiplying the result by 10 and taking 8 places after the decimal dot. This effectively yields 9 digits.
5. Mix the concatenation of the above inputs using a hash function, which can be either SHA1 or MD5. The **default is MD5**.

All the outputs in steps 1-4 are converted to string values using *sprintf*. The input buffer to the hash function is therefore a char array. Assuming the default MD5 function is used, **the result** is 128 bits session-id, which is converted to 32 ASCII characters.

Support for external entropy file: there is support for giving external entropy to the process, which is supplemented to the sources in steps 1-4 as another input to the hash function. The support is by specifying a value to the key *session.entropy_file* which is an entropy file source and *session.entropy_length* which is the amount of entropy to read from the file during the allocation process. One can use this setting by pointing it to /dev/[u]random for instance. The configuration is done by configuring these values in the *php.ini* runtime configuration file. **There isn't any default entropy source** and *session.entropy_length* defaults to 0 in the default PHP runtime configuration (as documented in http://php.net/manual/en/session.configuration.php). We do note that the implementation reads *MIN (2048, session.entropy_length)* bytes from the entropy file during the allocation process.

Following this, in the next analysis, we assume no external entropy source was defined.

In the next two sections we outline our attack on the session id allocation algorithm of PHP. Our attack is comprised of two steps: (a) Finding out the state of the generator using

some guess work and the attack we showed on the generator in the previous section, (b) utilizing the attack to effectively hijack a valid session-id.

Entropy injection during the allocation process: we note that the session id allocation algorithm is well thought and the developers didn't only rely on the security of the PRNG to allocate the session-id (like we saw in [42]). This can be seen in steps 1-3: the algorithm injects more entropy to the session id generation process, besides from the entropy of the PRNG itself.

10.3 Extracting the state of the generator

The attack requires getting 3 consecutive valid session-ids from the server. As we'll show, the effort required for extracting the generator's state is $\mathbf{2^{44}}$, which can be achieved using today's moderate machines.

We denote $\boldsymbol{X_n}$ as the buffer generated by steps 1-4 and we'll denote the session-id generated from this buffer using $\boldsymbol{f(X_n)}$, which is the invocation of the hash function. We assume that the default hash function, MD5 is used. However, we note that using SHA1 doesn't affect our attack at all. We further denote as $\boldsymbol{S(X_n)}$ the state of the generator *g* whose output was used in step 4.

The extraction attack outline follows: **First**, we perform 3 **consecutive queries** to the server in order to get 3 valid session-ids, $f(X_1)$, $f(X_2)$ and $f(X_3)$. Since we get valid session-ids originating from our machine, the remote-address adds no entropy, as it is known to us (it's our machine's IP). So the only entropy comes from the time parameters; we'll bound it and show that the space of X_n is small enough. **Next**, we perform an exhaustive search to find X_1 from $f(X_1)$, X_2 from $f(X_2)$ and X_3 from $f(X_3)$. From the X_1, X_2, X_3 we just found we get the corresponding 3 outputs of the generator. **Finally**, we extract the state $S(X_1), S(X_2), S(X_3)$ using the method outlined in section 8.2.3.1. We use the third output to validate our finding.

We'll now describe the attack in more details:

Guessing X_n: we follow with a detailed analysis of how many bits we need to guess in order to get to X_n.

1. **Remote Address** – as mentioned, we know this value.
2. **Time (seconds)** – we assume that the session-allocation procedure is one of the first operations that the server performs following a request. This assumption makes sense, as the session-id is probably needed in order to perform operations within the application. We further assume that we have the means to know the time of the server, to a seconds' accuracy, and the uncertainty of the request handling time is less than a second. This is reasonable if we consider a use of NTP by the server or getting the server time using other methods outlined in this paper. This means we have **0 entropy** from this parameter.
3. **Time (micro-seconds)** – we denote T_{micro} as the upper limit of the uncertainty, in micro-seconds, we allow ourselves to have regarding the time of the operation. This means that we need to guess $log_2(T_{micro})$ bits. We assume that in some conditions we can guess the time up to 5 milliseconds. This gives us an upper limit of $log_2(5{,}000) \cong \mathbf{12\ bits}$. This follows an assumption that that the server would take up to 1 msec to perform the actual session-id allocation from the instant the request reaches the server. The assumption of network latency uncertainty is actually 3-4 msec; this is from empirical data we gathered.
4. **Generator's output** – reminding ourselves that the output of the generator is a fraction between (0, 1) and the algorithm in step 4 takes only 9 digits from the generator's out-

put, we need to take into account rounding issues. Denote *epsilon=4.656613e-10*, *z* – the real generator's output, before multiplying it with epsilon, z' – a different generator's output that resulted in the same *output* that was taken in the algorithm. By examining the maximum difference between the two candidates, we see that we would need to test 3 situations. This follows from this observation:

z, z' go to the same value after the truncation =>

The corresponding fraction (before the truncation) will be the same up to the 9th digit =>

$$\text{epsilon} * |\, z - z' \,| < 10^{-9}$$

$$|\, z - z' \,| < 10^{-9}/\text{epsilon}$$

$$|\, z - z' \,| < 10^{-9} * 4.656613e\text{-}10$$

$$|\, z - z' \,| \leq 2$$

So there are at most 3 possibilities for the generator output given the truncated fraction.

To conclude: X_n has only 43 unknown bits, 12 from the microseconds and 31 from the generator output.

Extraction of the generator state effort: following the observations above, given an output $f(X_i)$ we would need to perform a search over a space of 2^{43} to find X_i (we'll enumerate all the 2^{43}). Since we do it for the three consecutive outputs, we get an effort of $3 * 2^{43}$.

Now, In order to get to $S(X_n)$ we now need to mount the attack we outlined in 8.2.3; the effort we needed there was $\sim 2^{17}$. Remembering that due to rounding issues we actually need to perform this 3 times per X_n yields an effort of $3 * 2^{17}$ per X_n. Since we perform the attack for two X_n and the third one is only used for verification gets us to a final effort of:

$$3 \cdot 2^{43} + 3^3 \cdot 2 \cdot 2^{17} \cong \mathbf{2^{44}}$$

Notes regarding attack assumptions and vectors:

1. **Consecutive outputs –** we are aware of the problematic assumption of getting consecutive outputs. We note that if we allow ourselves to only know the distance (amount of session ids generated) between the outputs we could still try to mount the attack above (In Attack 1 described in the next section we'll show a practical way to do so). However this might yield a more expensive attack to get the state of the generator, since the effort would not be 2^{17}, but $4 * {a_1}^k \leq 2^{31}$, where *k* is the amount of invocation/distance between the first two outputs and a_1 is the same as shown in 0. Another possible relaxation is to assume that we can bound the distance between the session-ids. E.g., if we were to know that we get the three sessions in a range of 1000 sessions, and then we can say that we have 1000 options for the difference between $f(X_1)$ and $f(X_2)$ and another 1000 for $f(X_2)$ and $f(X_3)$, thus getting 1,000,000 options.
2. **Attack the initial seed of the generator** - we note that breaking the initial seed is still applicable; however it's not as beneficial. Due to the fact that we will need to know how many invocations took place after it was generated to get ourselves synched with the current state of the generator.

10.4 Mounting the Session Hijacking Attack

In the previous section we've managed to get the state of the generator; since no new entropy is added to the generator itself and it poses no forward security we can now traverse to older or newer states as we please. We now continue to describe 3 online attacks

that would allow us to get to a valid session-id of a user and thus hijacking her session. The attacks are presented in a decreasing strength we allow our attacker to possess.

For the sake of convenience, we'll denote *eve* as the attacker and *bob* as the attacked user, whose session we hijack.

Attack 1: assuming *eve* is a passive MITM (man in the middle), thus she can **passively listen to all the traffic of the server**. In order to make the attack scenario realistic and non-trivial, we further assume that the communication of the server is encrypted using https [105].

Since *eve* listens to the traffic, she knows the originating IP (from a lower level protocol) of each request and the timing of the requests to a milli-second precision.

Denote the session expiry timeout time as T_{expiry}. *eve* monitors the traffic of the server for T_{expiry} and keeps track of the originating IPs for the requests during this period. After T_{expiry} *eve* assumes that each new request from an IP or an existing IP that hasn't accessed the server for T_{expiry} resulted in creation of a new session-id.

eve then requests 3 valid session-ids from her own machine and mounts the attack we described in the previous section to reveal the internal state of the PRNG. While mounting the attack *eve* continues to keep track on the amount of new session-ids created (note that even if the outputs are not consecutive eve knows the amount of PRNG executions).

Now *eve* continues to monitor the traffic of the server and **chooses** a new session-id, Sid_{bob} that was created to her liking. *eve* knows how many invocations of the PRNG were performed (amount of new session-ids created) since the time she revealed the generator's state. Furthermore, *eve* knows the time of Sid_{bob} to milli-second accuracy. The only unknown is the exact time of micro-second, within this milli-second that Sid_{bob} was created. To solve this *eve* tries 1000 values of micro-seconds to generate 1000 potential session-ids and sends to the server for validation. One is expected to be a valid session-id, which is the attack goal.

The above attack has a weakness that the server can suspect that an attack on *bob*'s session, actually *bob's* IP address, undergoes and can decide to stop serving requests to bob. To avoid this, we can **further improve our attack** as follows.

eve won't attack a specific user, but instead she will keep track of 1000 new session ids created by the server, say in 1000 seconds. As described above, for each such session creation the only uncertainty is in the exact creation time, which is in the order of 1000 micro-seconds. So to achieve **one valid** session-id it's enough for *eve* to iterate over the 1000 session-ids, guess the exact time of each session-id creation, construct a session-id accordingly and try to validate it by sending it to the server. In **expectance** this will yield a **single** valid session-id.

This variation has **two main advantages**: (a) we only send one invalid cookie per (faked) IP so our attack is more concealed, (b) in the previous scheme the server could protect itself from the attack by simply blocking the attacked IP (and by doing that the server will also be protected from our attack if we are attacking from several machines). However in this variation the server doesn't know which session-id we are going to attack, thus making it much harder for the server to protect itself.

Note: we assumed that we can monitor the number of session-ids created (and correspondingly, number of PRNG advancing) by listening to the secured https communication and following all the IPs of this communications. Using an IP from a lower level protocol in order to keep track of users that are **behind a proxy/NAT** can be problematic, since the IP sent would be of the NAT/proxy and not the actual user. This can be solved by using infor-

mation regarding side-channels of HTTPS recently covered by Schneier in [106] regarding the paper [107]. For instance, we can assume users that reach the homepage or login page would get a new session id. By using side-channels we can try and deduce that indeed the user is viewing the mentioned pages.

Note: as mentioned above, **involving the server** isn't the best way to validate our guess since the server can easily understand that an attack is performing and could delay responses or completely block our IP. Nevertheless, we expect that most applications wouldn't go to the trouble of theses pre-cautions and even if they will, we can always mount the attack from multiple machines, in order to conceal our attack.

Attack 2: assuming *eve* can only passively listen to all the traffic **between the server and *bob*.** The attack is similar to the attack proposed before; however we now have an uncertainty regarding the amount of new session-ids issued by the server.

Number of session ids: in order to try and predict the number of invocations, $N_{new\text{-}sessions}$, we will make an educated guess and involve the server in validating our guess. Our guess will be based on the assumption of the average amount of new sessions in a specific time frame. This assumption is application specific, as it translates from the expected load of a specific website. We will denote the lower bound of this guess as $N_{num\text{-}sessions}^{min}$ and the upper bound as $N_{num\text{-}sessions}^{max}$.

As before, we still have an uncertainty of 1000 micro-seconds regarding the exact time of the session id creation. Mounting the attack requires iterating over the values of 1000 * ($N_{num\text{-}sessions}^{max}$ - $N_{num\text{-}sessions}^{min}$) and handing the constructed session-ids to the server for validation.

Attack 3: the only assumption regarding *eve* is that she can **interact with the server**. As expected this attack requires the largest amount of guess work and interaction with the server in order to validate our guess.

Here we don't know a **concrete IP**, nor do we know of a valid IP. Nevertheless, we observe that the implementation only takes the REMOTE_ADDR and doesn't look into the X_FORWARDED_FOR header, which is a **bad practice that causes a weakness in the system** that the attacker can exploit as follows. Effectively, if the user is behind a proxy, the proxy's address will be sent as the REMOTE_ADDR. A lot of traffic is generated from parties behind a proxy server, mainly for organizations. Even worse, some proxies don't even send the REMOTE_ADDR header, thus yielding in an empty string for this parameter. We can conclude that for practical attacks, this isn't a real obstacle. We can always either target all the empty REMOTE_ADDR values for scenarios that the header wasn't used or target a known big corporation by getting its proxy's IP addresses.

From here on the attack continues exactly like described in Attack 2, by bounding the session-id exact creation time.

In conclusion: we showed concrete examples of attacks on the session-id generation in PHP that rely on the fact that not enough entropy is being injected when generating new sessions id and that we can relatively easily break the PRNG's state.

We note that in order to validate if we got a valid session-id we simply use some indication within the application, e.g., seeing the user-name or a shopping cart. This validation step is important as PHP simply generates a new session-id incase the session-id it got is invalid.

11 Appendix B: Code Snippets

11.1 Java

11.1.1 Java: SecureRandom

11.1.1.1 perm_table

```
56, 30, -107, -6, -86, 25, -83, 75, -12, -64,
5, -128, 78, 21, 16, 32, 70, -81, 37, -51,
-43, -46, -108, 87, 29, 17, -55, 22, -11, -111,
-115, 84, -100, 108, -45, -15, -98, 72, -33, -28,
31, -52, -37, -117, -97, -27, 93, -123, 47, 126,
-80, -62, -93, -79, 61, -96, -65, -5, -47, -119,
14, 89, 81, -118, -88, 20, 67, -126, -113, 60,
-102, 55, 110, 28, 85, 121, 122, -58, 2, 45,
43, 24, -9, 103, -13, 102, -68, -54, -101, -104,
19, 13, -39, -26, -103, 62, 77, 51, 44, 111,
73, 18, -127, -82, 4, -30, 11, -99, -74, 40,
-89, 42, -76, -77, -94, -35, -69, 35, 120, 76,
33, -73, -7, 82, -25, -10, 88, 125, -112, 58,
83, 95, 6, 10, 98, -34, 80, 15, -91, 86,
-19, 52, -17, 117, 49, -63, 118, -90, 36, -116,
-40, -71, 97, -53, -109, -85, 109, -16, -3, 104,
-95, 68, 54, 34, 26, 114, -1, 106, -121, 3,
66, 0, 100, -84, 57, 107, 119, -42, 112, -61,
1, 48, 38, 12, -56, -57, 39, -106, -72, 41,
7, 71, -29, -59, -8, -38, 79, -31, 124, -124,
8, 91, 116, 99, -4, 9, -36, -78, 63, -49,
-67, -87, 59, 101, -32, 92, 94, 53, -41, 115,
-66, -70, -122, 50, -50, -22, -20, -18, -21, 23,
-2, -48, 96, 65, -105, 123, -14, -110, 69, -24,
-120, -75, 74, 127, -60, 113, 90, -114, 105, 46,
27, -125, -23, -44, 64
```

11.2 .NET

11.2.1 System.Random (Random.cs)

The following is the code of the *System.Random* class as generated by the Reflector.

```
// ==++==
//
//
//    Copyright (c) 2006 Microsoft Corporation.  All rights reserved.
//
//    The use and distribution terms for this software are contained in
the file
//    named license.txt, which can be found in the root of this
distribution.
//    By using this software in any fashion, you are agreeing to be bound
by the
//    terms of this license.
//
//    You must not remove this notice, or any other, from this software.
//
//
// ==--==
/*============================================================
**
** Class:  Random
**
**
** Purpose: A random number generator.
**
**
```

```
==========================================================*/
namespace System {

    using System;
    using System.Runtime.CompilerServices;
    using System.Globalization;
[System.Runtime.InteropServices.ComVisible(true)]
    [Serializable()] public class Random {
      //
      // Private Constants
      //
      private const int MBIG =  Int32.MaxValue;
      private const int MSEED = 161803398;
      private const int MZ = 0;


      //
      // Member Variables
      //
      private int inext, inextp;
      private int[] SeedArray = new int[56];

      //
      // Public Constants
      //

      //
      // Native Declarations
      //

      //
      // Constructors
      //

      public Random()
        : this(Environment.TickCount) {
      }

      public Random(int Seed) {
        int ii;
        int mj, mk;

        //Initialize our Seed array.
        //This algorithm comes from Numerical Recipes in C (2nd Ed.)
        mj = MSEED - Math.Abs(Seed);
        SeedArray[55]=mj;
        mk=1;
        for (int i=1; i<55; i++) {  //Apparently the range [1..55] is
special (Knuth) and so we're wasting the 0'th position.
          ii = (21*i)%55;
          SeedArray[ii]=mk;
          mk = mj - mk;
          if (mk<0) mk+=MBIG;
          mj=SeedArray[ii];
        }
        for (int k=1; k<5; k++) {
          for (int i=1; i<56; i++) {
        SeedArray[i] -= SeedArray[1+(i+30)%55];
        if (SeedArray[i]<0) SeedArray[i]+=MBIG;
          }
        }
        inext=0;
        inextp = 21;
        Seed = 1;
      }

      //
      // Package Private Methods
      //
```

```
/*==================================Sample==============================
======
      **Action: Return a new random number [0..1) and reSeed the Seed
array.
      **Returns: A double [0..1)
      **Arguments: None
      **Exceptions: None

=========================================================================
====*/
      protected virtual double Sample() {
          //Including this division at the end gives us significantly
improved
          //random number distribution.
          return (InternalSample()*(1.0/MBIG));
      }

      private int InternalSample() {
          int retVal;
          int locINext = inext;
          int locINextp = inextp;

          if (++locINext >=56) locINext=1;
          if (++locINextp>= 56) locINextp = 1;

          retVal = SeedArray[locINext]-SeedArray[locINextp];

          if (retVal<0) retVal+=MBIG;

          SeedArray[locINext]=retVal;

          inext = locINext;
          inextp = locINextp;

          return retVal;
      }

      //
      // Public Instance Methods
      //



/*==================================Next================================
======
      **Returns: An int [0..Int32.MaxValue)
      **Arguments: None
      **Exceptions: None.

=========================================================================
====*/
      public virtual int Next() {
        return InternalSample();
      }

      private double GetSampleForLargeRange() {
          // The distribution of double value returned by Sample
          // is not distributed well enough for a large range.
          // If we use Sample for a range [Int32.MinValue..Int32.MaxValue)
          // We will end up getting even numbers only.

          int result = InternalSample();
          // Note we can't use addition here. The distribution will be bad
if we do that.
          bool negative = (InternalSample()%2 == 0) ? true : false;  //
decide the sign based on second sample
          if( negative) {
```

```
            result = -result;
        }
        double d = result;
        d += (Int32.MaxValue - 1); // get a number in range [0 .. 2 *
Int32MaxValue - 1)
        d /= 2*(uint)Int32.MaxValue - 1  ;
        return d;
    }



/*====================================Next==============================
======
    **Returns: An int [minvalue..maxvalue)
    **Arguments: minValue -- the least legal value for the Random
number.
    **           maxValue -- One greater than the greatest legal return
value.
    **Exceptions: None.

================================================================================
====*/
    public virtual int Next(int minValue, int maxValue) {
        if (minValue>maxValue) {
            throw new
ArgumentOutOfRangeException("minValue",String.Format(CultureInfo.CurrentCu
lture, Environment.GetResourceString("Argument_MinMaxValue"), "minValue",
"maxValue"));
        }

        long range = (long)maxValue-minValue;
        if( range <= (long)Int32.MaxValue) {
            return ((int)(Sample() * range) + minValue);
        }
        else {
            return (int)((long)(GetSampleForLargeRange() * range) +
minValue);
        }
    }



/*====================================Next==============================
======
    **Returns: An int [0..maxValue)
    **Arguments: maxValue -- One more than the greatest legal return
value.
    **Exceptions: None.

================================================================================
====*/
    public virtual int Next(int maxValue) {
        if (maxValue<0) {
            throw new ArgumentOutOfRangeException("maxValue",
String.Format(CultureInfo.CurrentCulture,
Environment.GetResourceString("ArgumentOutOfRange_MustBePositive"),
"maxValue"));
        }
        return (int)(Sample()*maxValue);
    }



/*====================================Next==============================
======
    **Returns: A double [0..1)
    **Arguments: None
    **Exceptions: None
```

```
==========================================================================
====*/
      public virtual double NextDouble() {
        return Sample();
      }


/*==================================NextBytes===============================
======
      **Action:  Fills the byte array with random bytes [0..0x7f].  The
entire array is filled.
      **Returns:Void
      **Arugments:  buffer -- the array to be filled.
      **Exceptions: None

==========================================================================
====*/
      public virtual void NextBytes(byte [] buffer){
        if (buffer==null) throw new ArgumentNullException("buffer");
        for (int i=0; i<buffer.Length; i++) {
          buffer[i]=(byte)(InternalSample()%(Byte.MaxValue+1));
        }
      }
    }


}
```

11.2.2 System.Security.Cryptography. RNGCryptoServiceProvider (rngcryptos-erviceprovider.cs)

```
// ==++==
//
//
//    Copyright (c) 2006 Microsoft Corporation.  All rights reserved.
//
//    The use and distribution terms for this software are contained in
the file
//    named license.txt, which can be found in the root of this
distribution.
//    By using this software in any fashion, you are agreeing to be bound
by the
//    terms of this license.
//
//    You must not remove this notice, or any other, from this software.
//
//
// ==--==

//
// RNGCryptoServiceProvider.cs
//

namespace System.Security.Cryptography {
    using Microsoft.Win32;
    using System.Runtime.InteropServices;

[System.Runtime.InteropServices.ComVisible(true)]
    public sealed class RNGCryptoServiceProvider : RandomNumberGenerator {

        //
        // public constructors
        //

        public RNGCryptoServiceProvider() { }
```

```
        //
        // public methods
        //

        public override void GetBytes(byte[] data) {
            if (data == null) throw new ArgumentNullException("data");
            if (!Win32Native.Random(true, data, data.Length))
                throw new
CryptographicException(Marshal.GetLastWin32Error());
        }

        public override void GetNonZeroBytes(byte[] data) {
            if (data == null)
                throw new ArgumentNullException("data");

            GetBytes(data);

            int indexOfFirst0Byte = data.Length;
            for (int i = 0; i < data.Length; i++) {
                if (data[i] == 0) {
                    indexOfFirst0Byte = i;
                    break;
                }
            }
            for (int i = indexOfFirst0Byte; i < data.Length; i++) {
                if (data[i] != 0) {
                    data[indexOfFirst0Byte++] = data[i];
                }
            }

            while (indexOfFirst0Byte < data.Length) {
                // this should be more than enough to fill the rest in one
iteration
                byte[] tmp = new byte[2 * (data.Length -
indexOfFirst0Byte)];
                GetBytes(tmp);

                for (int i = 0; i < tmp.Length; i++) {
                    if (tmp[i] != 0) {
                        data[indexOfFirst0Byte++] = tmp[i];
                        if (indexOfFirst0Byte >= data.Length) break;
                    }
                }
            }
        }
    }
}
```

11.2.2.1 Microsoft.Win32 (win32natives.cs)

Here are only the applicable parts of the code, not the entire class code:

```
[DllImport(KERNEL32, EntryPoint="PAL_Random")]
        [ResourceExposure(ResourceScope.None)]
        internal extern static bool Random(bool bStrong,
                                [Out, MarshalAs(UnmanagedType.LPArray)] byte[]
buffer, int length);
```

11.2.3 win32pal.c

Here are only the applicable parts of the code, not the entire code:

```
PALIMPORT
BOOL
PALAPI
PAL_Random(
        IN BOOL bStrong,
```

```
        IN OUT LPVOID lpBuffer,
        IN DWORD dwLength)
{
    BOOL Ret;
    HCRYPTPROV hProv;

    PERF_ENTRY(PAL_Random);
    LOGAPI("PAL_Random(bStrong=%d, lpBuffer=%p, dwLength=0x%x)\n",
bStrong, lpBuffer, dwLength);

    if (hCryptProv == NULL)
    {
        Ret = CryptAcquireContext(&hProv, NULL, NULL, PROV_RSA_FULL,
CRYPT_VERIFYCONTEXT);

        if (!Ret)
            goto LExit;

        if (InterlockedCompareExchangePointer((PVOID*)&hCryptProv,
(PVOID)hProv, NULL) != NULL)
        {
            // somebody beat us to it
            CryptReleaseContext(hProv, 0);
        }
    }

    Ret = CryptGenRandom(hCryptProv, dwLength, lpBuffer);

LExit:
    LOGAPI("PAL_Random returns BOOL %d\n", Ret);
    PERF_EXIT(PAL_Random);
    return Ret;
}
```

11.3 *NIX C

11.3.1 BSD

Configuration of PRNG variants: The *TYPE_0* is a special type – it falls back to using LCG as the PRNG. Each type uses a different polynomial (actually trinomial). The following is defined per type: (1) *BREAK_i* (where *i = 1..5*) - the minimum amount of state information (in bytes), from which this type (trinomial) is used, (2) *DEG_i* - the degree of the trinomial used (3) *SEP_i* – the separation between the two lower order of coefficients of the trinomial, meaning the separation between *fptr* and *rptr*. The types are defined as bellow:

```
/* Linear congruential.  */
#define     TYPE_0          0
#define     BREAK_0     8
#define     DEG_0       0
#define     SEP_0       0

/* x**7 + x**3 + 1.  */
#define     TYPE_1          1
#define     BREAK_1     32
#define     DEG_1       7
#define     SEP_1       3

/* x**15 + x + 1.  */
#define     TYPE_2          2
#define     BREAK_2     64
#define     DEG_2       15
#define     SEP_2       1

/* x**31 + x**3 + 1.  */
#define     TYPE_3          3
#define     BREAK_3     128
```

```
#define     DEG_3       31
#define     SEP_3       3

/* x**63 + x + 1.  */
#define     TYPE_4          4
#define     BREAK_4     256
#define     DEG_4       63
#define     SEP_4       1
```

LCG code for TYPE_0 in the BSD variant:

```
int32_t val = state[0];
val = ((state[0] * 1103515245) + 12345) & 0x7fffffff;
```

Generator code of the AFG generators; below is the code snippet of this algorithm: (line 383-405, random_r.c)

```
int32_t *fptr = buf->fptr;
int32_t *rptr = buf->rptr;
int32_t *end_ptr = buf->end_ptr;
int32_t val;

val = *fptr += *rptr;
/* Chucking least random bit.  */
*result = (val >> 1) & 0x7fffffff;
++fptr;
if (fptr >= end_ptr)
{
  fptr = state;
  ++rptr;
}
else
{
  ++rptr;
  if (rptr >= end_ptr)
    rptr = state;
}
buf->fptr = fptr;
buf->rptr = rptr;
```

The LCG that is used in the seed initialization process is in the following snippet (line 192-201, *random_r.c)*:

```
/* This does:
   state[i] = (16807 * state[i - 1]) % 2147483647;
   but avoids overflowing 31 bits.  */
long int hi = word / 127773;
long int lo = word % 127773;
word = 16807 * lo - 2836 * hi;
if (word < 0)
word += 2147483647;
```

11.3.2 SVID

drand structure:

```
struct drand48_data
  {
    unsigned short int __x[3];      /* Current state.  */
    unsigned short int __old_x[3]; /* Old state.  */
    unsigned short int __c;   /* Additive const. in congruential formula.
*/
    unsigned short int __init;      /* Flag for initializing.  */
    unsigned long long int __a;     /* Factor in congruential formula.  */
  };
```

The LCG implementation can be seen in the function *__drand48_iterate*; the code follows:

```
int
__drand48_iterate (xsubi, buffer)
     unsigned short int xsubi[3];
     struct drand48_data *buffer;
{
  uint64_t X;
  uint64_t result;

  /* Initialize buffer, if not yet done.  */
  if (__builtin_expect (!buffer->__init, 0))
    {
      buffer->__a = 0x5deece66dull;
      buffer->__c = 0xb;
      buffer->__init = 1;
    }

  /* Do the real work.  We choose a data type which contains at least
     48 bits.  Because we compute the modulus it does not care how
     many bits really are computed.  */

  X = (uint64_t) xsubi[2] << 32 | (uint32_t) xsubi[1] << 16 | xsubi[0];

  result = X * buffer->__a + buffer->__c;

  xsubi[0] = result & 0xffff;
  xsubi[1] = (result >> 16) & 0xffff;
  xsubi[2] = (result >> 32) & 0xffff;

  return 0;
}
```

12 Appendix C: Configuration Files

12.1 java.security default security file configuration

Below we can see the default configuration (as taken from a Windows JDK installation) that is shipped with the java JRE and JDK. The applicable keys are highlighted in red; these keys are used for the PRNG generation algorithms.

```
#
# This is the "master security properties file".
#
# In this file, various security properties are set for use by
# java.security classes. This is where users can statically register
# Cryptography Package Providers ("providers" for short). The term
# "provider" refers to a package or set of packages that supply a
# concrete implementation of a subset of the cryptography aspects of
# the Java Security API. A provider may, for example, implement one or
# more digital signature algorithms or message digest algorithms.
#
# Each provider must implement a subclass of the Provider class.
# To register a provider in this master security properties file,
# specify the Provider subclass name and priority in the format
#
#    security.provider.<n>=<className>
#
# This declares a provider, and specifies its preference
# order n. The preference order is the order in which providers are
# searched for requested algorithms (when no specific provider is
# requested). The order is 1-based; 1 is the most preferred, followed
# by 2, and so on.
#
# <className> must specify the subclass of the Provider class whose
# constructor sets the values of various properties that are required
# for the Java Security API to look up the algorithms or other
# facilities implemented by the provider.
#
# There must be at least one provider specification in java.security.
# There is a default provider that comes standard with the JDK. It
# is called the "SUN" provider, and its Provider subclass
# named Sun appears in the sun.security.provider package. Thus, the
# "SUN" provider is registered via the following:
#
#    security.provider.1=sun.security.provider.Sun
#
# (The number 1 is used for the default provider.)
#
# Note: Providers can be dynamically registered instead by calls to
# either the addProvider or insertProviderAt method in the Security
# class.

#
# List of providers and their preference orders (see above):
#
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=com.sun.net.ssl.internal.ssl.Provider
security.provider.4=com.sun.crypto.provider.SunJCE
security.provider.5=sun.security.jgss.SunProvider
security.provider.6=com.sun.security.sasl.Provider
security.provider.7=org.jcp.xml.dsig.internal.dom.XMLDSigRI
security.provider.8=sun.security.smartcardio.SunPCSC
security.provider.9=sun.security.mscapi.SunMSCAPI

#
# Select the source of seed data for SecureRandom. By default an
# attempt is made to use the entropy gathering device specified by
# the securerandom.source property. If an exception occurs when
# accessing the URL then the traditional system/thread activity
```

```
# algorithm is used.
#
# On Solaris and Linux systems, if file:/dev/urandom is specified and it
# exists, a special SecureRandom implementation is activated by default.
# This "NativePRNG" reads random bytes directly from /dev/urandom.
#
# On Windows systems, the URLs file:/dev/random and file:/dev/urandom
# enables use of the Microsoft CryptoAPI seed functionality.
#
securerandom.source=file:/dev/urandom
#
# The entropy gathering device is described as a URL and can also
# be specified with the system property "java.security.egd". For example,
#   -Djava.security.egd=file:/dev/urandom
# Specifying this system property will override the securerandom.source
# setting.

#
# Class to instantiate as the javax.security.auth.login.Configuration
# provider.
#
login.configuration.provider=com.sun.security.auth.login.ConfigFile

#
# Default login configuration file
#
#login.config.url.1=file:${user.home}/.java.login.config

#
# Class to instantiate as the system Policy. This is the name of the class
# that will be used as the Policy object.
#
policy.provider=sun.security.provider.PolicyFile

# The default is to have a single system-wide policy file,
# and a policy file in the user's home directory.
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy

# whether or not we expand properties in the policy file
# if this is set to false, properties (${...}) will not be expanded in
policy
# files.
policy.expandProperties=true

# whether or not we allow an extra policy to be passed on the command line
# with -Djava.security.policy=somefile. Comment out this line to disable
# this feature.
policy.allowSystemProperty=true

# whether or not we look into the IdentityScope for trusted Identities
# when encountering a 1.1 signed JAR file. If the identity is found
# and is trusted, we grant it AllPermission.
policy.ignoreIdentityScope=false

#
# Default keystore type.
#
keystore.type=jks

#
# Class to instantiate as the system scope:
#
system.scope=sun.security.provider.IdentityDatabase

#
# List of comma-separated packages that start with or equal this string
# will cause a security exception to be thrown when
# passed to checkPackageAccess unless the
```

```
# corresponding RuntimePermission ("accessClassInPackage."+package) has
# been granted.
package.access=sun.,com.sun.xml.internal.ws.,com.sun.xml.internal.bind.,co
m.sun.imageio.

#
# List of comma-separated packages that start with or equal this string
# will cause a security exception to be thrown when
# passed to checkPackageDefinition unless the
# corresponding RuntimePermission ("defineClassInPackage."+package) has
# been granted.
#
# by default, no packages are restricted for definition, and none of
# the class loaders supplied with the JDK call checkPackageDefinition.
#
#package.definition=

#
# Determines whether this properties file can be appended to
# or overridden on the command line via -Djava.security.properties
#
security.overridePropertiesFile=true

#
# Determines the default key and trust manager factory algorithms for
# the javax.net.ssl package.
#
ssl.KeyManagerFactory.algorithm=SunX509
ssl.TrustManagerFactory.algorithm=PKIX

#
# The Java-level namelookup cache policy for successful lookups:
#
# any negative value: caching forever
# any positive value: the number of seconds to cache an address for
# zero: do not cache
#
# default value is forever (FOREVER). For security reasons, this
# caching is made forever when a security manager is set. When a security
# manager is not set, the default behavior is to cache for 30 seconds.
#
# NOTE: setting this to anything other than the default value can have
#       serious security implications. Do not set it unless
#       you are sure you are not exposed to DNS spoofing attack.
#
#networkaddress.cache.ttl=-1

# The Java-level namelookup cache policy for failed lookups:
#
# any negative value: cache forever
# any positive value: the number of seconds to cache negative lookup
results
# zero: do not cache
#
# In some Microsoft Windows networking environments that employ
# the WINS name service in addition to DNS, name service lookups
# that fail may take a noticeably long time to return (approx. 5 seconds).
# For this reason the default caching policy is to maintain these
# results for 10 seconds.
#
#
networkaddress.cache.negative.ttl=10

#
# Properties to configure OCSP for certificate revocation checking
#

# Enable OCSP
#
```

```
# By default, OCSP is not used for certificate revocation checking.
# This property enables the use of OCSP when set to the value "true".
#
# NOTE: SocketPermission is required to connect to an OCSP responder.
#
# Example,
#   ocsp.enable=true

#
# Location of the OCSP responder
#
# By default, the location of the OCSP responder is determined implicitly
# from the certificate being validated. This property explicitly specifies
# the location of the OCSP responder. The property is used when the
# Authority Information Access extension (defined in RFC 3280) is absent
# from the certificate or when it requires overriding.
#
# Example,
#   ocsp.responderURL=http://ocsp.example.net:80

#
# Subject name of the OCSP responder's certificate
#
# By default, the certificate of the OCSP responder is that of the issuer
# of the certificate being validated. This property identifies the
certificate
# of the OCSP responder when the default does not apply. Its value is a
string
# distinguished name (defined in RFC 2253) which identifies a certificate
in
# the set of certificates supplied during cert path validation. In cases
where
# the subject name alone is not sufficient to uniquely identify the
certificate
# then both the "ocsp.responderCertIssuerName" and
# "ocsp.responderCertSerialNumber" properties must be used instead. When
this
# property is set then those two properties are ignored.
#
# Example,
#   ocsp.responderCertSubjectName="CN=OCSP Responder, O=XYZ Corp"

#
# Issuer name of the OCSP responder's certificate
#
# By default, the certificate of the OCSP responder is that of the issuer
# of the certificate being validated. This property identifies the
certificate
# of the OCSP responder when the default does not apply. Its value is a
string
# distinguished name (defined in RFC 2253) which identifies a certificate
in
# the set of certificates supplied during cert path validation. When this
# property is set then the "ocsp.responderCertSerialNumber" property must
also
# be set. When the "ocsp.responderCertSubjectName" property is set then
this
# property is ignored.
#
# Example,
#   ocsp.responderCertIssuerName="CN=Enterprise CA, O=XYZ Corp"

#
# Serial number of the OCSP responder's certificate
#
# By default, the certificate of the OCSP responder is that of the issuer
# of the certificate being validated. This property identifies the
certificate
```

```
# of the OCSP responder when the default does not apply. Its value is a
string
# of hexadecimal digits (colon or space separators may be present) which
# identifies a certificate in the set of certificates supplied during cert
path
# validation. When this property is set then the
"ocsp.responderCertIssuerName"
# property must also be set. When the "ocsp.responderCertSubjectName"
property
# is set then this property is ignored.
#
# Example,
#   ocsp.responderCertSerialNumber=2A:FF:00
```

13 Bibliography

[1] Donald E Knuth, *Seminumerical Algorithms*, 2nd ed.: Addison-Wesley, 1997, vol. 2, The Art of Computer Programming Series.

[2] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes in C (The Art of Scientific Computing)*, 2nd ed.: CAMBRIDGE UNIVERSITY PRESS, 1992.

[3] TIOBE Software. (2010, August) TOIBE Software. [Online]. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[4] Hex-Rays. The IDA Pro Disassembler and Debugger. [Online]. http://www.hex-rays.com/idapro/

[5] Boaz Barak and Shai Halevi, "An architecture for robust pseudo-random generation," , 2005, http://www.cs.princeton.edu/~boaz/Papers/devrand.pdf.

[6] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall, "Cryptanalytic Attacks on Pseudorandom Number Generators," *Lecture Notes in Computer Science*, vol. 1372, pp. 168-188, 1998.

[7] M. Gude, "Concept for a High-Performance Random Number Generator Based on Physical Random Noise," vol. 39, pp. 187-190, 1985.

[8] G. B Agnew, "Random Source for Cryptographic Systems," in *Advances in Cryptology - EUROCRYPT '87 Proceedings*, 1988, pp. 77-81.

[9] M. Richterm, *Ein Rauschgenerator zur Gweinnung won quasi-idealen Zufallszahlen fur die stochastische Simulation*.: Aachen University of Technology, 1992, In German.

[10] R.C. Fairchild, R.L. Mortenson, and K.B. Koulthart, "An LSI Random Number Generator (RNG)," in *Advances in Cryptology: Proceedings*, 1985, pp. 203-230.

[11] Jon Postel, "Transmission control protocol," *Internet Engineering Task force*, vol. RFC 793, September 1981.

[12] INTERNATIONAL TELECOMMUNICATION UNION (X.667), "Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 object identifier components," X.667, 2004.

[13] Tim Dierks and Christopher Allen, "The TLS protocol version 1.0," RFC 2246, 1999.

[14] R.L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, vol. 21 (2), pp. 120–126, 1978.

[15] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*.: CRC Press, 1996.

[16] Shamir A., "On the generation of cryptographically strong pseudo-random sequences," , 1981, pp. 544-550.

[17] Lenore Blum, Manuel Blum, and Michael Shub, "Comparison of two pseudo-random numbergenerators," , New York, 1983, pp. 61-78.

[18] Michael Howard and David LeBlanc, *Writing Secure Code, Second Edition*.: Microsoft Publishing, 2002.

[19] D. H. Lehmer, "Mathematical methods in large-scale computing units," in *2nd Sympos. on Large-Scale Digital Calculating Machinery*, Cambridge, MA, 1949, pp. 141-146.

[20] L'ecuyer Pierre, "Efficient and Portable Combined Random Number Generators," *Communications of the ACM*, vol. 31 Number 6, no. June 1988, 1988.

[21] Bruce Schneier, *Applied Cryptography, Second Edition: Protocols, Algorthms, and Source Code in C*.: John Wiley & Sons, Inc., 1996.

[22] Marsaglia George, "A Current View of Random Number Generators," , Atlanta, 1984.

[23] Richard P. Brent, "Uniform Random Number Generators for Supercomputers," , Melbourne, 1992.

[24] Lewis T. G. and Payne W. H., "Generalized Feedback Shift Register Pseudorandom Number Algorithm," vol. 20, pp. 456-468, 1973.

[25] Matsumoto M. and Kurita Y., "Twisted GFSR generators," *ACM Transactions on Modeling and Computer Simulation*, vol. 2, pp. 179-194, 1992.

[26] M. Matsumoto and Y. Kurita, "Twisted GFSR generators II," *ACM Transactions on Modeling and Computer Simulation*, vol. 4, pp. 254-266, 1994.

[27] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator," *ACM Trans-actions on Modeling and Computer Simulation (TOMACS)*, vol. 8, pp. 3-30, 1998.

[28] National Institute of Standards and Technology, "NIST 800-90: Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised)," NIST, NIST NIST 800-90, 2007.

[29] FIPS, "DIGITAL SIGNATURE STANDARD (DSS)," FIPS PUB 186-2, 2000.

[30] National Institute of Standards and Technology, "FIPS PUB 197: Advanced Encryption Standard (AES)," FIPS PUB 197, 2001.

[31] Dan Shumow and Ferguson Niels. On the Possibility of a Back Door in the NIST SP800-90 Dual Ec Prng. [Online]. http://rump2007.cr.yp.to/15-shumow.pdf

[32] National Institute of Standards and Technology, "FIPS PUB 180-1: Secure hash standard," FIPS PUB 180-1, 1995.

[33] National Bureau of Standards, "FIPS-Pub.46: Data Encryption Standard," Washington D.Cc, FIPS-Pub.46, 1977.

[34] Mihir Bellare, Shafi Goldwasser, and Daniele Micciancio, ""Pseudo-Random" Number Generation within Cryptographic Algorithms: the DSS Case," in *Advances in Cryptology - Crypto 97 Proceedings*, 1997.

[35] Peter Gutmann, "Software Generation of Practically Strong Random Numbers," in *In Proc. of 7th USENIX Security Symposium*, 1998, An updated version appears in http://www.cypherpunks.to/~peter/06_random.pdf.

[36] Ernesto Guisado. Cryptographic Random Numbers. [Online]. http://erngui.com/rng/index.html

[37] GEORGE MARSAGLIA, "RANDOM NUMBERS FALL MAINLY IN THE PLANES," 1968.

[38] George A Fishman and Louis R III Moore, "An exhaustive analysis of multiplicative congruential random number generators with modulus 2^31-1," *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 1, pp. 24-45, 1986.

[39] Ian Goldberg and David Wagner, "Randomness and the Netscape Browser," *Dr. Dobb's Journal*, 1996.

[40] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller, "Kerberos: an authentication service for open network systems," in *Proc. winter Usenix conference*, Dallas, 1988.

[41] Nelson Minar, "Breakable session keys in Kerberos v4," message-ID 199602200828.BAA21074@nelson.santafe.edu, 1996.

[42] Zvi Gutterman and Dahlia Malkhi, "Hold Your Sessions: An Attack on Java Session-Id Generation," , 2005.

[43] Apache Software Foundation (ASF). Tomcat Server.

[44] M. E. Hellman, "A cryptanalytic time-memory trade off," *IEEE Trans. Inform. Theory*, pp. 401-406, 1980.

[45] Wietse Venema, "Murphy's law and computer security," in *Proceedings of the 6th Usenix Security Symposium*, 1996, p. 187.

[46] Brad Arkin et al. (1999, September) How we Learned to Cheat in Online Poker: A Study in Software Security. [Online]. http://www.cigital.com/papers/download/developer_gambling.pdf

[47] ISO/SEC, "Pascal," ISO 7185 :1990, 1991.

[48] Top 500 Supercomputer Sites. (2010, June) Top 500 Supercomputer Sites. [Online]. http://top500.org/stats/list/35/osfam

[49] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman, "Analysis of the Linux Random Number Generator," , 2006.

[50] Ted Ts'o. random.c. [Online]. http://www.kernel.org

[51] Open WRT Platform. [Online]. http://www.openwrt.org

[52] (2010) Operating Systems Market Share. [Online]. http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=10

[53] Leo Dorrendorf, Zvi Gutterman, and Benny Pinkas, "Cryptanalysis of the Windows Random Number Generator," 2007.

[54] Gregg Keizer. (2007, November) Microsoft confirms that XP contains random number generator bug. [Online]. http://www.computerworld.com/s/article/9048438/Microsoft_confirms_that_XP_contains_random_number_generator_bug

[55] Microsoft. Microsoft Windows Homepage. [Online]. http://www.microsoft.com/windows/

[56] Linux. Linux. [Online]. http://www.linux.org/

[57] Microsoft. Microsoft's CLR Overview. [Online]. http://msdn.microsoft.com/en-us/library/ddk909ch.aspx

[58] GNU. GNU C Library. [Online]. http://www.gnu.org/software/libc/

[59] GNU. BSD Random Number Functions. [Online]. http://www.gnu.org/software/libc/manual/html_node/BSD-Random.html

[60] GNU. SVID Random Number Function. [Online]. http://www.gnu.org/s/libc/manual/html_node/SVID-Random.html

[61] Microsoft. rand_s API. [Online]. http://msdn.microsoft.com/en-us/library/sxtz2fa8(VS.80).aspx

[62] Ilpo Vattulainen, T. Ala-Nissila, and K. Kankaala, "Physical Tests for Random Numbers in Simulations," 1994.

[63] Wikipedia. C POSIX Library. [Online]. http://en.wikipedia.org/wiki/C_POSIX_library

[64] Microsoft. Microsoft C Run-Time Libraries. [Online]. http://msdn.microsoft.com/en-us/library/abx4dbyh(VS.80).aspx

[65] ISO/IEC. Open Standards. [Online]. http://www.open-std.org/jtc1/sc22/wg14/www/standards.html#9899

[66] Wikipedia. ANSI C. [Online]. http://en.wikipedia.org/wiki/ANSI_C

[67] ANSI. ANSI C Rationale Document. [Online]. http://www.lysator.liu.se/c/rat/d10.html#4-10-2

[68] Joan B. Plumstead, "Inferring a Sequence Produced by a Linear Congruence," , 1982.

[69] Microsoft. Security Enhancements in the CRT. [Online]. http://msdn.microsoft.com/en-us/library/8ef0s5kh(v=VS.80).aspx

[70] Wikipedia. CryptGenRandom Using RtlGenRandom. [Online]. http://en.wikipedia.org/wiki/CryptGenRandom#Using_RtlGenRandom

[71] Microsoft. Security-Enhanced Versions of CRT Functions. [Online]. http://msdn.microsoft.com/en-us/library/wd3wzwts(v=VS.80).aspx

[72] Wikipedia. iOS. [Online]. http://en.wikipedia.org/wiki/IPhone_OS

[73] GNU. GLIBC Pseudo-Random Numbers API. [Online]. http://www.gnu.org/s/libc/manual/html_node/Pseudo_002dRandom-Numbers.html#Pseudo_002dRandom-Numbers

[74] Klein Amit. (2008, February) PowerDNS Recursor DNS Cache Poisoning. [Online]. http://www.trusteer.com/list-context/publications/powerdns-recursor-dns-cache-poisoning

[75] Wikipedia. System V Interface Definition. [Online]. http://en.wikipedia.org/wiki/System_V_Interface_Definition

[76] Sun (now Oracle). Oracle and Java Technologies. [Online]. http://www.oracle.com/us/technologies/java/index.html

[77] Jan P. Monsch. Iplosion. [Online]. http://www.iplosion.com/papers/ruining_security_with_java.util.random_v1.0.pdf

[78] FIPS, Security Requirements for Cryptographic Modules, FIPS 140-2, 2001, http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf.

[79] D. Eastlake 3rd and S. D. Crocker and J. Schiller, "Randomness Recommendations for Security," RFC 1750, 1994.

[80] Sun. Java Cryptography Architecture. [Online]. http://download.oracle.com/javase/1.4.2/docs/guide/security/CryptoSpec.html

[81] Microsoft. Cryptographic Service Providers. [Online]. http://msdn.microsoft.com/en-us/library/aa380245(VS.85).aspx

[82] Tzachy Reinman and Malkhi Dahlia. (2005) On Linux Random Number Generator Thesis Dissertation. [Online]. www.cs.huji.ac.il/~reinman/thesis.pdf

[83] RSA Laboratories, "PKCS #11: Cryptographic Token Interface Standard,".

[84] Brian Warner. EGD: The Entropy Gathering Daemon. [Online]. http://egd.sourceforge.net/

[85] E. H. McKinney, "Generalized Birthday Problem," *American Mathematical Monthly*, pp. 385-387, 1966.

[86] X. Wang, Y.L. Yin, and and H. Yu, "Finding Collisions in the Full SHA-1," , 2005.

[87] Microsoft, "Common Language Infrastructure (CLI)," ECMA Standard ECMA-335,.

[88] Microsoft. (2006, March) Shared Source Common Language Infrastructure 2.0 Release. [Online]. http://www.microsoft.com/downloads/en/details.aspx?FamilyID=8c09fd61-3f26-4555-ae17-3121b4f51d4d&displaylang=en

[89] redgate..NET Reflector - redgate products. [Online]. http://www.red-gate.com/products/reflector/

[90] Novell. mono. [Online]. http://www.mono-project.com/Main_Page

[91] Wikipedia. The Golden Ratio. [Online]. http://en.wikipedia.org/wiki/Golden_ratio

[92] David Wright. Random Numbers. [Online]. http://www.shadlen.org/ichbin/random/generators.htm#knuth

[93] Microsoft. (2007, September) MSDN (.NET Matters). [Online]. http://msdn.microsoft.com/en-us/magazine/cc163367.aspx

[94] Microsoft. Mapping Algorithm Names to Cryptography Classes. [Online]. http://msdn.microsoft.com/en-us/library/693aff9y(v=VS.90).aspx

[95] Michael Howard. (2005, January) Cryptographically Secure Random number on Windows without using CryptoAPI. [Online]. http://blogs.msdn.com/b/michael_howard/archive/2005/01/14/353379.aspx

[96] Microsoft. (2010, July) CryptGenRandom Function API Documentation. [Online]. http://msdn.microsoft.com/en-us/library/aa379942(VS.85).aspx

[97] PHP. PHP. [Online]. http://www.php.net/

[98] Netcraft. PHP Usage. [Online]. http://www.php.net/usage.php

[99] Apache. Apache. [Online]. http://www.apache.org/

[100] PHP. PHP Writing Functions Documentation. [Online]. http://www.php.net/manual/en/internals2.funcs.php

[101] Ingo Molnar. (2020, September) lockless, scalable get_pid(). [Online]. http://lwn.net/Articles/10181/

[102] corbet. (2002, September) Solving the process ID allocation problem. [Online]. http://lwn.net/Articles/10238/

[103] corbet. (2002, September) The get_pid() function. [Online]. http://lwn.net/Articles/10246/

[104] Redhat. RedHat Linux. [Online]. http://www.redhat.com/

[105] Wikipedia. HTTP Secure. [Online]. http://en.wikipedia.org/wiki/HTTP_Secure

[106] Bruce Schneier. (2010, March) Side-Channel Attacks on Encrypted Web Traffic. [Online]. http://www.schneier.com/blog/archives/2010/03/side-channel_at.html

[107] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang, "Side-Channel Leaks in Web Applications: a Reality Today, a Challenge Tomorrow," , Oakland, 2010.

[108] Firewall.cx. Firewall.cx, the site for Networking Professionals. [Online]. http://www.firewall.cx/dns-query-format.php

[109] J. Kohl and C. Neuman, "The Kerberos network authentication service (V5)," RFC 1510, 1993.

[110] B. Callaghan, B. Pawlowski, and P. Staubach, "NFS version 3 protocol specification," RFC 1813, 1995.

תקציר

לעתים קרובות מפתחי תוכנה נדרשים לשלב מספרים אקראים במערכות וביישומים אותן הן מפתחים. המערכות והיישומים נעים מתחומים כגון מימוש פרוטוקול אבטחה חדש ועד מימוש אלגוריתם ערבוב קלפים עבור משחק פוקר מקוון. שפות תוכנה מודרניות מגיעות עם ספריות תוכנה עשירות; ספריות תוכנה אלו גם מכילות פונקציות ומימושים של מחוללי מספרים פסאודו-אקראיים. פונקציות אלו מפשטות את עבודת המפתח ומאפשרות לו לחולל מספרים פסאודו אקראיים ללא הצורך בכתיבת המחוללים בעצמו. המימושים השונים נבדלים בתכונות שונות, כגון החוזק הקריפטוגרפי וכן באלגוריתם הייצור.

בעבודה זו אנו חוקרים את המימושים של מחוללי מספרים פסאודו-אקראיים בשפות תכנות פופולריות. אנו מספקים ניתוח מפורט ושלם של האלגוריתמים באותם מימושים, חוזקם הקריפטוגרפי והיכולות של אותם מחוללים. אנו מראים חולשות רבות במחוללים הנ"ל, לרבות באג במימוש אחד המחוללים בשפת התכנות #C וכן התקפה לא טריוויאלית על אלגוריתם ייצור הסשן ב PHP. ההתקפה מתבססת על ניתוח וחולשה אשר מצאנו באחד המחוללים הקיימים ב PHP.

המרכז הבינתחומי בהרצליה
בית-ספר אפי ארזי למדעי המחשב

מחולליי מספרים פסאודו-אקראיים בשפות תכנות

מוגש כחיבור סופי לפרויקט מחקר תואר מוסמך

על-יד אביב סיני

העבודה בוצעה בהנחיית ד"ר צבי גוטרמן (Cloudshare, האונ' העברית)

מרץ 2011