Universidade Federal de Pernambuco
Centro de Informática

Predicting the Output of a Popular Unorthodox Mode of Use for C#'s System.Random

Lucas Miranda Lin

Advised by
Paulo André da Silva Gonçalves

September 3, 2021

Dedication

First and foremost to God, to the good, to the one from which all stems. Next, in no particular order: to my family, to my friends, and to my masters. I mention no names in the interest of anonymity and brevity; but if you are reading this, you are most likely in one of these groups. Know that, however I may appear to you, I extend my truest gratitude for that which you have shared with me.

List of Figures

Contents

1 Introduction

Computer programs need random numbers for many purposes: statistical testing, machine learning, gaming, algorithms, and cryptography are some of these. The task of procuring random numbers is left to a class of algorithms called Pseudorandom Number Generators (PRNGs). These produce apparently random numbers — yet in fact, they are completely deterministic.

To ease the work of programmers, most languages come imbued with a default PRNG implementation — a quick, easy-to-use generator, which produces numbers of randomness with sufficient quality for general applications. These are a good fit for most situations where random numbers are needed. However, special care must be taken wherever security mingles.

Most PRNGs are insecure, and the numbers they produce can often be predicted by a resourceful outside observer. This is not a problem for most use cases, but in some it is a disaster. For example, if a website generates authentication cookies with an insecure PRNG, a malicious user might be able to impersonate another.

A class called Cryptographically Secure PRNGs (CSPRNGs) is available for such scenarios, as these are designed to offer strong guarantees of unpredictability. But software development may take place under tight time constraints, and security is often sidelined in favour of other concerns. Thus it is commonplace for PRNGs to be found in use in places where they should not.

1.1 Motivation and Objectives

In this work, we set our sights upon **System.Random**, the default PRNG for the C# programming language. This generator is insecure, and a trivial prediction method is known for its conventional use case. We aim at another use case, unorthodox yet popular, which we call our **target use case**. A search on github returns tens of thousands of results for this mode of use in public repositories [1]. We have verified that at least some of these take place in security-critical contexts. Nonetheless the target use case has remained unstudied — there is no known algorithm capable of performing its prediction.

The aim of our research is to fill in this gap: to analyse System.Random so as to present a prediction method for the target use case. We seek to develop a predictor tool implementing our findings — a System.Random prediction tool attacking the target use case. Finally, we intend to test this tool and show our results so as to verify our methods.

1.2 Results

We introduce the ***NRN(seed)*** formula (Lemma 1), which describes the number produced by ***new Random(seed).Next()***. We combine this with C#'s default seed selection schemes to introduce prediction methods for the target use case, i.e. ***new Random().Next()***. Prediction is deterministic and unerring if the target generator runs on .NET Core, .NET 5, or Mono; and of varying accuracy if it runs on .NET Framework, depending on the quality of timing information available.

1.3 Document Overview

Excluding introduction and conclusion, our work is organised into four major sections:

- Section 2: an overview of pseudorandom number generation in general;
- Section 3: an overview of the target generator, System.Random;
- Section 4: an explanation of our methods to predict the *target use case*;
- Section 5: an introduction and test results for our prediction tool.

2 Pseudorandom Number Generators

> Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin.
>
> John von Neumann [2]

In this section, we cover the fundamentals of pseudorandom number generators: what they are, why they are necessary, what they aim to accomplish, how they operate in a general sense, and how one might evaluate a generator's quality for a certain application domain.

2.1 On the General Workings and Quality of a Generator

As suggests Neumann, it is an oxymoron to produce truly random numbers by deterministic means alone. However, it is possible to produce merely **apparent** randomness: numbers which might seem not to follow any obvious pattern, distributed in a uniform fashion. These are known as **pseudorandom numbers**, and algorithms designed to produce such numbers are called pseudorandom number generators (**PRNGs**).

To this end, PRNGs make use of simple bitwise or modular arithmetic operations, chosen carefully so that their output *appears* independent from their input - ergo random. Their algorithms thus seek to produce apparent randomness by repeated applications of these operations.

Implementation-wise, some internal state s is held and operated over, either a single integer or an array of integers. Initial values for this state are derived from a single *seed* integer as $f(seed)$. To generate a number, the internal state is modified with $s \leftarrow g(s)$, applying the operations described in the previous paragraph. Finally, a number is derived from the new state as $h(s)$ and returned. Figure 1 illustrates this scheme.

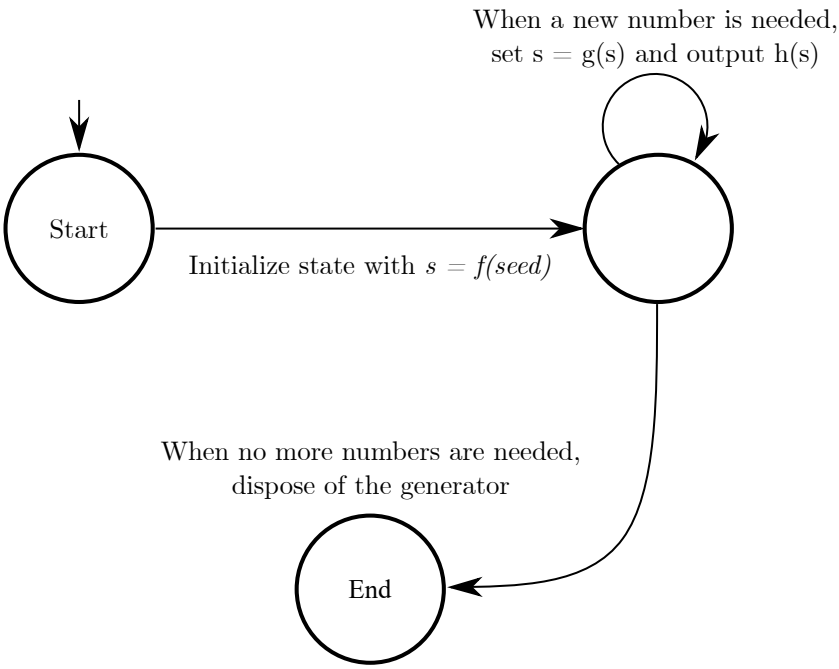

Figure 1: The basic, general operation of a PRNG

These algorithms are often implementations of mathematically defined recursive sequences. For instance, the *linear congruential generator* (LCG), a historically popular and well-studied PRNG [3], is defined by the sequence in Equation (1).

$$s_{i+1} = (s_i * a + b) \mod m \tag{1}$$

Different values for the constants a, b, and m define different instances of this generator; and s_0 is taken from the seed value to initiate the sequence.

In the literature, a generator's quality is most often argued for by a mathematical analysis of its sequence. A key property of a generator is its *period length*, meaning how many numbers are output before a PRNG's sequence begins to repeat itself. Too short a period implies insufficient randomness for a majority of applications, since a short and repeating cycle is wholly unconvincing.

Every PRNG must have *some* finite period length. As they base their algorithm off of some finite internal state, it follows that unlimited applications of g will eventually lead to some already-seen state configuration. From there, the generator proceeds to repeat itself, as shown in Figure 2.

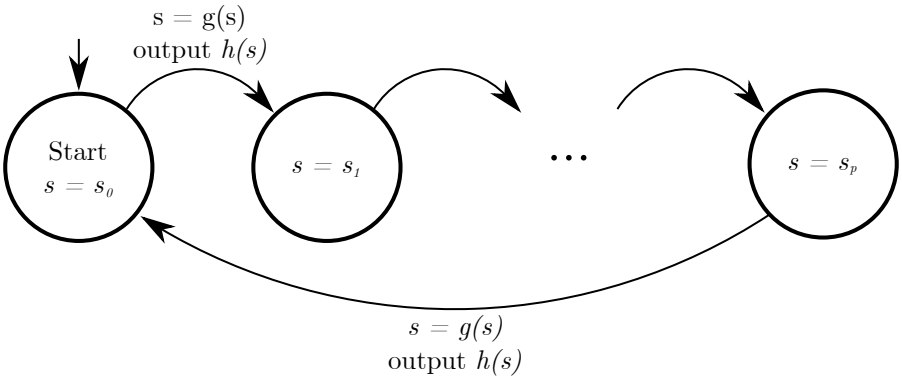

Figure 2: A generator with period p, as $g(s_p) = s_0$

A decent PRNG should have a large period. It has been shown [4] that an LCG with well-chosen constants a, b, and m has a period of m; often 2^{32}, as this is a convenient number for modular arithmetic. The more recent Mersenne Twister [5] has a gargantuan period length of $2^{19937} - 1$. System.Random's period is of unknown length — see 3.1 for more details.

Another attempt to discern the quality of PRNGs is to perform *statistical tests* on their output. These are designed to attempt to differentiate pseudorandom sequences from truly random numbers, employing a range of statistical tools to this end. A couple of popular programs that group, run, and report the results of many such tests are DIEHARD [6] and TESTU01 [7]. A generator is said to *pass* if these tests fail to distinguish their output from a truly random stream of numbers.

2.2 Cryptographic Security

Cryptography oft needs random numbers - keys, nonces, salts, and padding data are expected to be randomly generated. In cryptography the idea of *apparent* randomness is insufficient. Instead, generated numbers must be realistically unpredictable to an attacker; lest a key be leaked by an exploit targeting key generation, and thus an encryption broken. Typical PRNGs fail this requirement, and many are trivially predictable.

To attain unpredictability, **truly** random numbers would be ideal. Our best efforts towards this consist in measuring verily unpredictable physical phenomena, e.g. cosmic background radiation [8] or quantum events [9]. Even these may yet be argued philosophically to not be *truly* random, but they are at least humanly unpredictable, and therefore sufficient for cryptographic purposes.

Of course, most machines cannot use such expensive sources of randomness. Instead, they collect entropy from simpler phenomena, such as thermal noise [10], which are still unpredictable to an attacker. This task is often left to special hardware dedicated to this purpose, termed Hardware Random Number Generators (**HRNGs**) or True Random Number Generators (**TRNGs**). For example, the x64 ISA offers the **RDSEED** instruction [11], which on supported CPUs will provide a number sourced from a TRNG.

A key caveat of TRNGs is their slowness — they are often unable to provide random numbers at a fast enough rate to satisfy the needs of modern applications. This

is because even with simpler sources, entropy collection is still slow and expensive. Thus despite providing excellent randomness, TRNGs do not suffice by themselves.

Cryptographically Secure Pseudorandom Number Generators (**CSPRNGs**) step in to solve this performance issue. These are a subset of PRNGs, operating with the same concepts of seed and state, and are again deterministic algorithms which produce merely *apparently* random numbers. As such, they need not wait for entropy collection, and can produce many numbers as quickly as they can be ran.

However, to be deemed secure and suitable for cryptographic purposes, CSPRNGs are subject to great scrutiny [12, 13]. *Within polynomial computational time*, their output must be indistinguishable from a stream of truly random numbers, and an attacker must be unable to predict any future output by observing previous output. The latter is often extended to the reverse, so that any past output may also not be inferred by observing later output.

To meet these requirements, CSPRNGs must first pass all known statistical tests. The NIST Statistical Test Suite (STS) [14] was created for this purpose — it is *necessary* for any CSPRNG to pass these tests; however, this alone is not *sufficient*. As the NIST STS authors themselves posit, only cryptanalysis provides a strong guarantee of a CSPRNG's security. This means a generator's algorithm must be either proven to be unpredictable in polynomial time, or withstand all attempts by cryptanalysts to find polynomial-time attacks against its generation.

If a PRNG meets these requirements, it is deemed sufficient for cryptographic purposes - a CSPRNG. Although the numbers produced from this class of generators are not nigh impossible to predict, as are numbers from a TRNG, they are unpredictable *in polynomial time*.

CSPRNGs are seeded by the output of a TRNG, so that their seed is itself also secure - for if the seed were compromised, so too would the generator itself. Furthermore, they are periodically re-seeded, again with a TRNG. This means that *even if* an attack cracks a CSPRNG, this will not last for long, as it will soon refresh its state with new entropy. Figure 3 outlines the entire generation process for CSPRNGs.

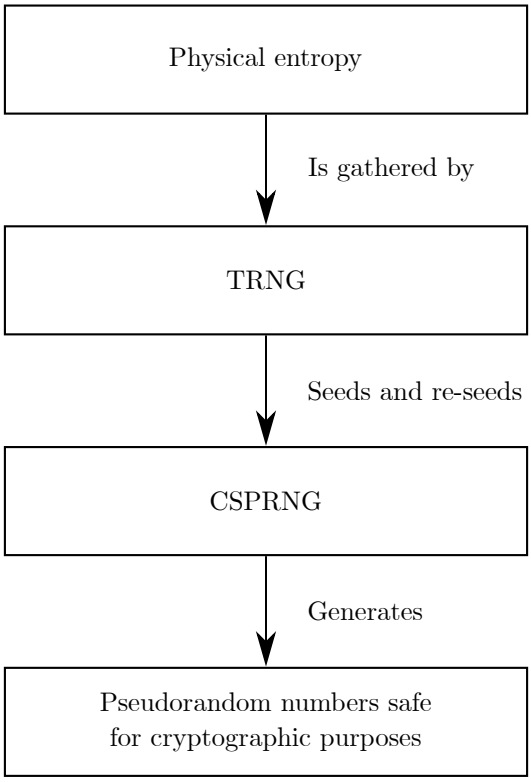

Figure 3: A summary of how cryptographically secure numbers are generated.

Although this class of generators is eponymously built with cryptography in mind, their design is such that they befit any use case wherein security is involved. For instance, a CSPRNG could safely be used to generate secret passcodes for a two-factor authentication mechanism. If instead a typical PRNG were used for such a mechanism, it would be left unsafe and open to prediction attacks and exploitation.

2.3 Conclusion

We describe the motivation for PRNGs, their basic operation, and how they may be graded as to the quality of their generation. We furthermore offer an overview of the motivation and overall operation behind CSPRNGs, and what standards they must meet in order to suit their intended purposes.

3 System.Random

The **Random** class, defined in the **System** namespace in C#, is this language's default PRNG. In this section, we first provide an overview of this generator; next, we explain how its output can be predicted. Finally, we introduce an unconventional mode of use, the **target use case** mentioned in 1.1, which drives this work.

3.1 Technical Overview

System.Random is a general-purpose PRNG, intended for end-user programmers. As such, its design focuses on ease-of-use and computationally inexpensive generation, yet maintaining enough randomness quality for generic usage.

To use this generator, programmers are expected to create and store an instance thereof by calling one of its constructors, and subsequently call its number-generating public methods according to their needs. We present below a list of these constructors and methods and a brief description for each.

- *new Random(int seed)*: Initialises an instance with the specified seed.
- *new Random()*: Initialises an instance with a default seed value.

- *int Next()*: Returns an integer in $[0, 2^{32} - 1[$.
- *int Next(int maxValue)*: Returns an integer in $[0, maxValue[$.
- *int Next(int minValue, int maxValue)*:
  Returns an integer in$[minValue, maxValue[$.
- *double NextDouble()*: Returns a double in $[0, 1[$.
- *void NextBytes(byte[] buffer)*: Fills *buffer* with random bytes.
- *void NextBytes(Span<byte> buffer)*: Fills *buffer* with random bytes.

Internally, all these methods source their randomness from a single private method ***InternalSample()***. This method returns an integer in $[0, 2^{32} - 1[$, and modifies the generator's internal state. We present another list below, describing how each public method uses the values returned by *InternalSample* to implement its desired behaviour.

- *int Next()*: Returns the value produced by *InternalSample()* with no alterations.
- *int Next(int maxValue)*: Returns
  $\lfloor maxValue \cdot InternalSample() \div (2^{32} - 1) \rfloor$
- *int Next(int minValue, int maxValue)*: Returns
  $\lfloor minValue + (maxValue - minValue) \cdot InternalSample() \div (2^{32} - 1) \rfloor$

- *double NextDouble()*: Returns *InternalSample()* $\div (2^{32} - 1)$
- *void NextBytes(byte[] buffer)*: For each byte b in *buffer*, overwrites b with the 8 least significant bits of *InternalSample()*. Note that *InternalSample()* will be called a total of n times if *buffer* has n bytes.
- *void NextBytes(Span<byte> buffer)*: Same as above.

The *InternalSample()* method is responsible for implementing System.Random's underlying PRNG algorithm. Its behaviour is such that for any single instance of System.Random, its nth call to *InternalSample()* will produce the $(n+55)$th call of the sequence described in Equation (2).

$$X_i = (X_{i-55} - X_{i-24}) \mod m, \quad m = 2^{32} - 1 \tag{2}$$

With $(X_0, ..., X_{54})$ derived from the seed value during initialization

This is called a *Lagged Fibonacci Sequence*, and a PRNG following such a sequence is a *Lagged Fibonacci Generator* (LFG). The numbers 55 and 24 are called its *lags*, and are of key relevance for the quality of its generation.

This version of the sequence with these lags was first studied as a candidate for PRNG algorithms by Mitchell and Moore [15], in an unpublished work known to us via Knuth [3]. They found that it features the desirable property of a very long period, if m is well-chosen. A period of at least $2^{55} - 1$ would be guaranteed if m were even, with a better yet guarantee of $(2^{55} - 1) \cdot 2^{e-1}$ if $m = 2^e$. However, no such guarantees are present for System.Random due to its writers' unfortunate choice of the odd number $m = 2^{32} - 1$.

Nevertheless, a 2020 publication by Hegadi and Patil [16] found good results for this PRNG in the NIST Statistical Test Suite [14] when compared to default generators from other languages, suggesting that despite the poor choice of m, System.Random is still a decent source for pseudo-randomness. Furthermore, various works [17, 18, 19] describe the implementation, in C#, of algorithms requiring some randomness. The authors do not explicitly specify how they produced this randomness, but it is reasonable to assume that it was sourced from System.Random, and that its generation was considered to be of sufficient quality.

Microsoft's code for this class was heavily inspired by *Numerical Recipes in C* [20], itself inspired by Knuth [3]. In the former, the authors include a sample implementation of the sequence in Equation (2) in the C programming language. We present a pseudocode algorithm mostly equivalent to this book's and System.Random's implementations in Algorithm 1.

Algorithm 1 System.Random equivalent pseudocode

```
1: procedure INIT                                  ▷ Initialise the generator...
2:     X ← [X_0, X_1, ..., X_54]                   ▷ ...with the first 55 values of (2)
3:     i ← 0
4:     j ← 21
5: end procedure
6: procedure SAMPLE                ▷ Returns the next number of the sequence
7:     X[i] ← (X[i] − X[j])  mod (2^32 − 1)
8:     x ← X[i]
9:     i ← (i + 1)  mod 55
10:    j ← (j + 1)  mod 55
11:    return x
12: end procedure
```

In summary, this algorithm stores $X_{n-1}, ..., X_{n-55}$ in an array of 55 integers. When the **SAMPLE** procedure is called, it computes X_n as $X_{n-55} - x_{n-24}$, which are stored in the array at $X[i]$ and $X[j]$, respectively. Then, it overwrites $X[i] = X_{n-55}$ with X_n; as the former will no longer be needed to compute any further elements in the sequence. Finally, X_n is returned.

This pseudocode will **not** output X_0 through X_{54}. Rather, the first call to SAMPLE will return X_{55}, the second will return X_{56}, and so on — this matches System.Random's behaviour. Also, we omit how the initial values $X_0, X_1, ..., X_{54}$ are chosen — they are derived from a single integer seed value, in a process we will later detail in 4.1.

Furthermore, System.Random is **not** a thread-safe PRNG. This means that if distinct threads were to call the *InternalSample()* method of one same instance at the same time, undefined behaviour may take place.

Although at a glance this may seem desirable (what could be more random than *undefined behaviour*?), this can also backfire and leave its internal state stuck in an undesirable configuration, leading to a catastrophic drop in the quality of its randomness. To quote Coveyou, "Random number generation is too important to be left to chance" ([21]). Thus, programmers who wish to use this class in a multithreaded context must take care to enforce its safe access.

3.2 Predicting the Common Case

Again, *System.Random* is **not** a CSPRNG. As C#'s default PRNG, its focus lies in general-purpose applications, where security is not a concern. We now show how an attacker may exploit its algorithm so as to predict its output.

We assume the attacker may observe as many samples as needed output by an instance of the target generator. Recall that in the SAMPLE procedure in algorithm 1, each sample is output by the generator and its value is then stored back into the its state array X. This means that by observing 55 consecutive samples, as in Figure 4, we will learn the entirety of the algorithm's internal state. Therewith, the target generator can be simply copied, i.e. by calling procedure INIT in Algorithm 1 and setting $X \leftarrow [S_0, S_1, ..., S_{54}]$. The copy's output will then be identical to the target's.

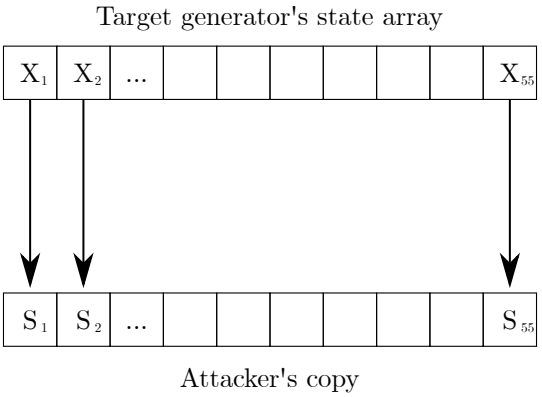

Figure 4: Capturing 55 samples, the attacker copies the target generator

This prediction method is trivial, as it follows immediately from System.Random's algorithm. It is a *state attack*, because its central theme is to uncover and reproduce the target generator's internal state to then imitate its behaviour.

Although it has not been discussed in any formal literature, many blog posts and forums around the web have described and demonstrated this attack. Furthermore, Krawczyk [22] and Boyar [23] circa 1990 published prediction methods targeting a very large class of PRNG algorithms, which happens to include System.Random. These authors' methods are less efficient than the attack described above, as they make less assumptions about their target generators — whereas their prediction might make a polynomial amount of mistakes, the method above makes none.

Prediction methods for all other number-generating System.Random methods (e.g. *NextFloat()*) may be derived from this attack; because all methods derive their generation from *InternalSample()*, and the *Next()* method simply returns a raw value from *InternalSample()*. We maintain this idea throughout our work, and describe only the prediction of the *Next()* method.

3.3 The Target Case

The previous subsection describes an attack against the common use case of System.Random, the way its authors meant for it to be used. In this subsection, we describe a different and somewhat improper use case which happens to escape that attack. This is our work's **target use case**.

Those who intend to use System.Random are expected to create and store a single instance of the class, and call number-generating methods from there. An alternative is to initialise and call a method from a whole new instance whenever a number is needed — this is condensed into the code *new Random().Next()*.

Re-initializing a generator for every number needed seems to only add unnecessary complexity, so why would a programmer opt to do this? The best advantage offered by this artifice is that it can lead to simpler code. To demonstrate this, we present a *SimpleRandom* static class, which handles System.Random's construction and wraps calls to its *Next()* method. We show two implementations of this class: the first of which uses a single instance of System.Random, and the second of which creates a new instance for each number that must be generated.

Listing 1: SimpleRandom - common case version

```
static class SimpleRandom
{
        private static Random r = new Random();

        public static int NextWrapper()
        {
                return r.Next();
        }
}
```

Listing 2: SimpleRandom - target case version

```
static class SimpleRandom
{
        public static int NextWrapper()
        {
                return new Random().Next();
        }
}
```

Despite the underlying complexity present in constructing a new System.Random instance for every call to *NextWrapper()* in Listing 2, its code is arguably simpler. Furthermore, recall from 3.1 that System.Random is **not** thread-safe. C# does provide a variety of threading tools which could be applied to Listing 1 so as to make its version of the *NextWrapper()* method thread-safe, at the cost of some extra code complexity. However, Listing 2 is **already** thread-safe, as distinct threads calling of *NextWrapper()* will each create their own distinct instance of System.Random.

But using the class in this manner has consequences. Whereas the numbers produced by calls to *NextWrapper()* in listing 1 will as expected follow sequence 2, the same does not hold for listing 2 — which instead follows a sequence determined by how the instances are initialised, and how their seeds are chosen.

Thus, our target case is this: **a program which generates pseudorandom numbers using System.Random, by calling *new Random().Next().*** To elucidate its relevance, we ran a github code search for "new Random().Next", and found tens of thousand results [1]. This suggests the target use case is not at all uncommon, and indeed the simplicity of its code is attractive to many programmers.

3.4 Conclusion

We cover a programmer's view of *System.Random*, and its underlying algorithm as a subtractive generator. We show how to predict the class's common use case through a state attack, and introduce and justify the target use case - where each time a number is needed, a new instance is created, taking only the first number it generates.

4 Predicting the Target Case

In this section, we analyse numbers generated by the target use case, as described in 3.3 and therein exemplified by Listing 2. We then leverage our findings to describe algorithms capable of predicting the target case. Furthermore, we consider what impediments a real-world prediction scenario might present, and adapt our algorithms to these.

Most or all of the analysis presented in this section was obtained by reading the source code [24, 25] and experimenting with System.Random. To the best of our knowledge, it is a novelty.

4.1 Problem Statement and Initialization Analysis

To predict the target use case as we have described is to predict the value of the following C# expression:

```
new Random().Next()
```

This expression does two things:

1. *new Random()*: Initialise a new instance of System.Random with a default seed value, selected by the executing .NET environment;
2. *.Next()*: Get the first number generated by this instance.

We will first look into what number is generated by *new Random(s).Next()*, for a known seed s. The initial state $X \leftarrow [X_0, ..., X_{54}]$ is derived from s by an algorithm introduced by a C implementation included in *Numerical Recipes in C* [20]. This was later translated to C# from C and placed into System.Random's code.

This algorithm is *very* convoluted, to a degree that impedes straightforward analysis. We encourage readers interested in its details to consult either the System.Random source code [24, 25] or *Numerical Recipes in C* [20]. In the book, the authors leave a comment regarding their code:

> *Now initialise the rest of the table, in a slightly random order, with numbers that are not especially random.* [20]

Thus the authors were aware their complicated code might regardless display regular behaviour, as indeed it does. If given a seed $\mathbf{s}$, this algorithm always produces an initial state $X \leftarrow [X_0, ..., X_{54}]$ such that:

$$X_{55} = a \cdot |s| + b \mod m$$

With constants a, b, m :
$a = 1121899819$
$b = 1559595546$
$m = 2^{32} - 1$

Recall that the first call to *Next()* from a new instance of System.Random returns X_{55}. We will hereafter refer to this as $NRN(s)$, an acronym for *new Random(s).Next()*:

Lemma 1.

$$NRN(s) = new\ Random(s).Next() = a \cdot |s| + b \mod m$$

$$\begin{aligned} &With\ constants\ a, b, m: \\ &a = 1121899819 \\ &b = 1559595546 \\ &m = 2^{32} - 1 \end{aligned}$$

Proof. The proof proceeds by brute force. In Appendix B, we show the source code for a C# program which proves that $NRN(s)$ = new Random(s).Next() for every 32-bit integer except $-(2^{32})$. □

To the best of our knowledge, we are the first to report on it this behaviour. It is accidental — no code overtly computing equation (1) nor either values a or b are present in the source code. We instead found these by observing the return of calls to *new Random(s).Next()* for incrementing values of s.

Similar formulae exist for X_{56}, X_{57}, and so on. These differ from equation (1) only by switching constants a and b for other values. As an example, we show below a formula for X_{56}:

$$X_{56} = a' \cdot |s| + b' \mod m$$

$$\begin{aligned} &\text{With constants } a', b', m: \\ &a' = 1755192844 \\ &b' = 1517371964 \\ &m = 2^{32} - 1 \end{aligned}$$

Appendix A lists such values for $X_{55}, X_{56}, ..., X_{109}$. The code in Appendix B can be modified to verify each of these. With equations for these 55 numbers of the sequence as a starting point, any X_n may then be found through equation (2).

Although our analysis and prediction target X_{55}, the *first* number output by a System.Random instance; since similar formulae exist for every X_n, our work may be converted to target any X_n, or equivalently any i-th output from System.Random.

4.1.1 Seed Selection and .NET Environments

Next, we look into how the seed value is chosen. Again, when calling *new Random().Next()*, the constructor with no arguments means that the seed s will be selected by the .NET environment itself. This is done in one of two ways:

1. On all versions of **.NET Framework**, $s \leftarrow$ ***Environment.TickCount***. This is a C# variable which references the environment's system time, as measured in milliseconds;

2. on **Mono**, **.NET 5** and all versions of **.NET Core**, seed values are pseudo-randomly generated by internal static instances of System.Random itself.

These are both commonplace patterns. For instance, Java's default PRNG class combines both techniques in its default seed selection [26]. We will inspect and propose prediction algorithms based on each .NET seed selection method in turn.

4.2 .NET Framework

4.2.1 Effecting a Prediction

According to its documentation, *Environment.TickCount* is a 32-bit signed integer referencing its machine's system timer, counting the amount of milliseconds elapsed since system startup. Its resolution thus depends on the system timer's, typically around 10 to 16 ms. If the system stays up for long enough (circa 25 days), this counter can overflow past its maximum value and into the negatives.

The most obvious attack route is to find the target machine's system timer's value s_t at the time of sample generation — the sample's value could then be calculated as $NRN(s_t)$. However, we present an alternative wherein only the *difference* between s_t and another sample's seed must be found.

Let there be a **target** sample with unknown value n_t, generated with a seed s_t; and a **base** sample with value n_b generated with seed s_b. Thus $NRN(s_t) = n_t$ and $NRN(s_b) = n_b$. Assume that only the values of n_b and the seed difference $\Delta s = s_t - s_b$ are known, but both s_t or s_b themselves are unknown. By Lemma 2, n_b and Δs are sufficient to calculate the value of n_t.

Lemma 2. $n_t = n_b + k \cdot a \cdot \Delta s \mod m$

$$
\begin{aligned}
&\textit{With } a, m, k: \\
&a = 1121899819 \\
&m = 2^{32} - 1 \\
&k = sgn(s_b) = sgn(s_t)
\end{aligned}
$$

Proof.

$$
\begin{aligned}
& n_b + k \cdot a(\Delta s) \mod m \\
=\ & n_b + k \cdot a(s_t - s_b) \mod m \\
=\ & NRN(s_b) + k \cdot a(s_t - s_b) \mod m \\
=\ & (a \cdot |s_b| + b) + k \cdot a(s_t - s_b) \mod m && \text{[Lemma 1]} \\
=\ & (a \cdot sgn(s_b) \cdot s_b + b) + k \cdot a(s_t - s_b) \mod m && \text{[Lemma 1]} \\
=\ & (a \cdot k \cdot s_b) + k \cdot a(s_t - s_b) + b \mod m \\
=\ & (a \cdot k \cdot s_b) + k \cdot a \cdot s_t - (k \cdot a \cdot s_b) + b \mod m \\
=\ & k \cdot a \cdot (s_t) + b \mod m \\
=\ & NRN(s_t) && \text{[Lemma 1]} \\
=\ & n_t
\end{aligned}
$$

□

This proof applies only if $sgn(s_b) = sgn(s_t)$. A negligible restriction, since *Environment.TickCount* only changes sign once every 25 days. Nevertheless, care must be taken if the target and base samples were generated at very distant times.

Also, the attacker knows not whether $k = 1$ or $k = -1$. There are two ways to handle this — one is to calculate $n_b + k \cdot a(\Delta s) \mod m$ twice, once for each value of k. One of these will necessarily result in the correct value for n_t.

Alternatively, suppose the attacker has access to one extra base sample with known value n'_b and seed difference $\Delta s' = s'_b - s_b$. Then the correct value for k may be discerned, as calculating $n_b + k \cdot a(\Delta s') \mod m$ will yield exactly n'_b only for the correct value of k. Again, since *Environment.TickCount* changes sign infrequently, it the same k can be assumed to hold for the target sample.

Finally, the largest difficulty is to actually find Δs. The attacker may *approximate* this value if time measurements t_t and t_b are available respectively for when n_t and n_b were generated: $t_t - t_b = \Delta t \approx \Delta s$. Herein lies this method's advantage over attempts to uncover s_t itself — the time measurements t_t and t_b need not be performed by the target machine's system timer, thus it needs not be compromised.

Yet Δt is only an approximation for Δs, since *Environment.TickCount* is itself an inaccurate, low-resolution clock. Thus, we must consider various **candidate values** for n_t, with $n_b + k \cdot a(\Delta t + \epsilon) \mod m$), using all integer values $e \leq \epsilon \leq e$ with e as the maximum relative error between the system timer and the clock responsible for Δt. The correct value for n_t is *guaranteed* to be among these candidates, and the attacker needs only test each of them to find it.

4.2.2 Practical Hindrance: Lag

Consider a PRNG attack scenario where the attacker sends HTTP requests over the internet to a server, which is programmed to generate a number with *new Random().Next()* upon receiving such requests. We shall analyse this context and show how Δs may best be approximated therein.

Let req_i be the time at which a request was sent, n_i be the pseudorandom number generated, and res_i be the time at which the corresponding response was received. In this scenario, the attacker is no longer privy to the exact time measurements t_b and t_t. Rather, he knows for sure only that $req_b \leq t_b \leq res_b$ and $req_t \leq t_t \leq res_t$. Even still, the attacker may compute bounds for all possible candidates for Δs.

Lemma 3. $req_t - res_b - e \leq \Delta s \leq res_t - req_b + e$

Proof. Since all valid candidates for t_t and t_b fall within $[req_t, res_t]$ and $[req_t, res_t]$ respectively, $t_t - t_b = \Delta t$ must be in $[req_t - res_b, res_t - req_b]$. Recall that all possible values for Δs are in $[\Delta t - e, \Delta t + e]$. Then Δs is in $[req_t - res_b - e, res_t - req_b + e]$. □

This range for Δs can be visualised in Figure 5: its correct value is guaranteed to fall somewhere in $[a - e, b + e]$. This interval can include *hundreds* of options, and thus incurs a severe penalty to the prediction's accuracy.

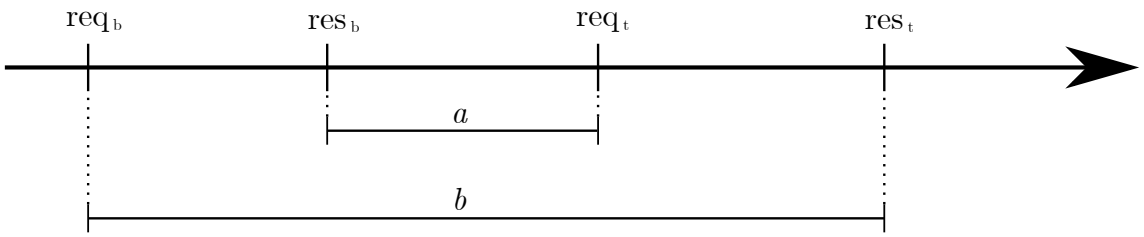

Figure 5: Two example requests for pseudorandom numbers

Some combinations for req_i, res_i and t_i are likelier than others. It is in the attacker's best interests to make an effort to take this into account, so as to estimate the *likeliest* values for Δs. The attacker would then prioritise these, e.g. by testing them for correctness first. This minimizes the inaccuracy introduced by the large range for Δs.

We consider that absent any further information, the *most likely* correct value for t_i is $\frac{1}{2}(res_i + req_i)$, with probabilities dropping proportionally as candidate values become more distant. Figure 6 illustrates this idea. We base this off an assumption that both request and response use the same network link, and thus take similar amounts of time to reach their respective targets.

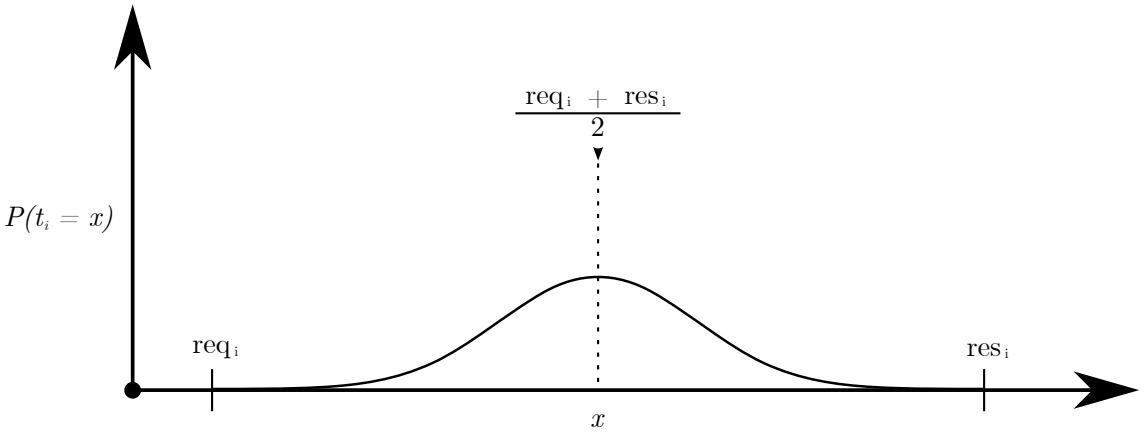

Figure 6: A possible distribution of candidate values for t_i.

4.2.3 Alternative Vectors

A few simpler alternatives to this attack are available — if the target server presents one of a few properties that cannot be affirmed generally. These would need to be verified, analysed and exploited in a case-by-case basis.

If the server happens to leak some time measurement performed server-side, this may be used by the attacker as a *precise* measurement for the time at which a request was processed by the server. This gives an exact value for t_i, thus dispenses with the need to manipulate req_i and res_i. If the time measurements come from the *system timer* itself, all complications can be skipped by computing $NRN(s_t)$.

A *synchronous generation* attack is available if the server can be induced to generate the base and target samples at *nearly* the same time. If the system timer

does not change value between them, then these two samples will share the exact same value.

4.3 .NET Core and Others

4.3.1 Effecting a Prediction

In all .NET environments besides .NET Framework, the default seed selection for System.Random is performed by internal, static instances of System.Random itself. These are two: s_globalRandom and t_threadRandom.

s_globalRandom is a single, global instance of System.Random. A program will only initialise it once, and it is seeded with the output of a CSPRNG. t_threadRandom is initialised once *per thread* — it is thread-static, meaning each thread will use its own separate instance of t_threadRandom. Each thread's instance is seeded by a call to *s_globalRandom.Next()*. Finally, each call to *new Random()* will be seeded by a call to its thread's *t_threadRandom.Next()*.

To summarise: each call to *new Random()* is seeded by its thread's t_threadRandom, which is seeded by the program-wide s_globalRandom, which is seeded by a CSPRNG. Figure 7 illustrates this process.

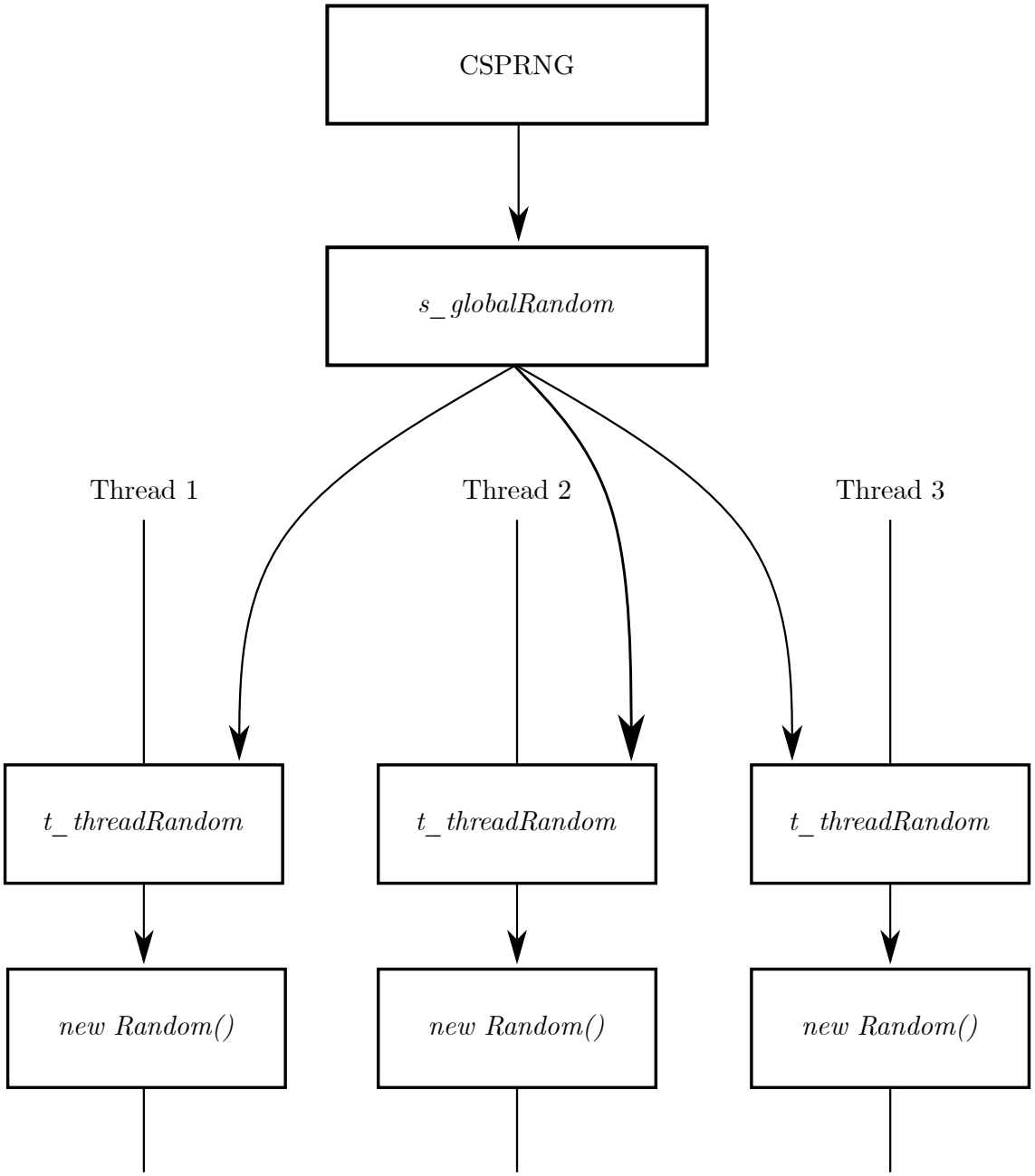

Figure 7: The *hierarchy* for .NET Core seed selection

Access to t_threadRandom and s_globalRandom is controlled to be thread-safe.

As this scheme's behaviour depends on how the target application makes use of threads, we describe how a prediction attack against the target use case may be performed in two different threading scenarios.

4.3.1.1 The Single-Threaded Case

Suppose **all** relevant calls to *new Random().Next()* take place on a **single** thread. Thus, a single instance is responsible for seeding all calls to *new Random()*.

Of course, t_threadRandom is itself an instance of *System.Random*, and as such would be vulnerable to the common case attack as described in 3.2. However, the attacker cannot observe t_threadRandom's raw output. As illustrates Figure 8, each T_i generated by this instance is consumed by *new Random().Next()*, and the attacker may only observe $NRN(T_i)$.

Figure 8: Only $NRN(T_i)$ can be observed by the attacker - T_i is never revealed.

But a property of *NRN* opens an alternative prediction method. Suppose $NRN(T_{i-55})$ and $NRN(T_{i-24})$ are observed. Note that per System.Random's generating sequence (2), $T_i = T_{i-55} - T_{i-24} \mod m$.

Lemma 4. $NRN(T_i) = NRN(T_{i-55}) - NRN(T_{i-24}) + b \mod m$

$$\begin{aligned} &\textit{With constants } b, m: \\ &b = 1559595546 \\ &m = 2^{32} - 1 \end{aligned}$$

Proof.

$$\begin{aligned} & NRN(T_{i-55}) - NRN(T_{i-24}) + b \mod m \\ &= a \cdot T_{i-55} + b - (a \cdot T_{i-24} + b) + b \mod m && \text{[Lemma 1]} \\ &= a \cdot T_{i-55} + b - a \cdot T_{i-24} - b + b \mod m \\ &= a \cdot (T_{i-55} - T_{i-24}) + b \mod m \\ &= a \cdot (T_i) + b \mod m \\ &= NRN(T_i) && \text{[Lemma 1]} \end{aligned}$$

□

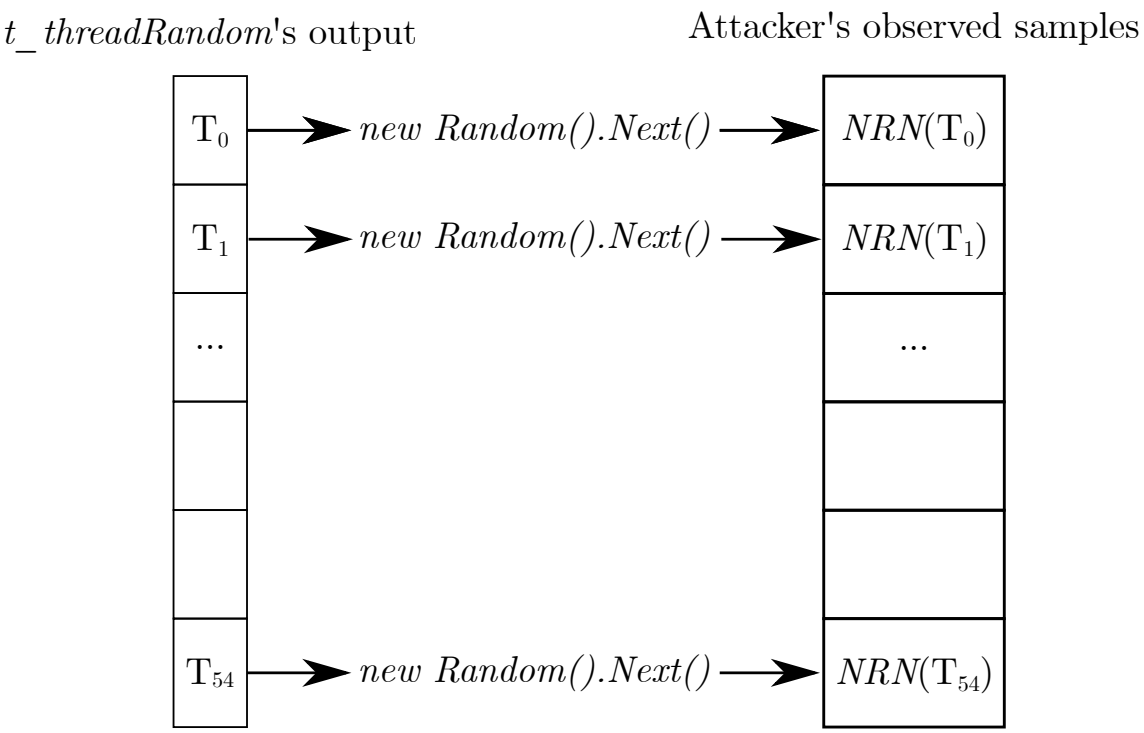

Figure 9: 55 consecutive values for $NRN(T_i)$ are sufficient to mount an attack

An attack backed by Lemma 4 looks much like the common case attack as in 3.2. The attacker would observe 55 **consecutive** samples, which represent $NRN(T_0)$ through $NRN(T_{54})$, as in Figure 9. Therewith, all future and past output $NRN(T_i)$ can be calculated with $NRN(T_i) = NRN(T_{i-55}) - NRN(T_{i-24}) + b \mod m$; as per Lemma 4.

4.3.1.2 The Always-New Case

Suppose the polar opposite of the single-threaded case: all relevant calls to *new Random().Next()* now take place on a new thread. In this case, each thread will only make a *single* call to *new Random().Next()*. Thus each thread will consume one number generated by s_globalRandom to seed its t_threadRandom, and grab *only* the first number generated by the latter to finally seed *new Random()*.

Suppose a seed value s_i is produced by s_globalRandom. Then this usage of t_threadRandom is equivalent to a call to *new Random(s_i).Next()* — creating a new instance seeded with s_i and grabbing only its first output. Thus the programmer's call to *new Random()* is seeded with $NRN(s_i)$, and the output of *new Random().Next()* will be $NRN(NRN(s_i))$, as shown in Figure 10.

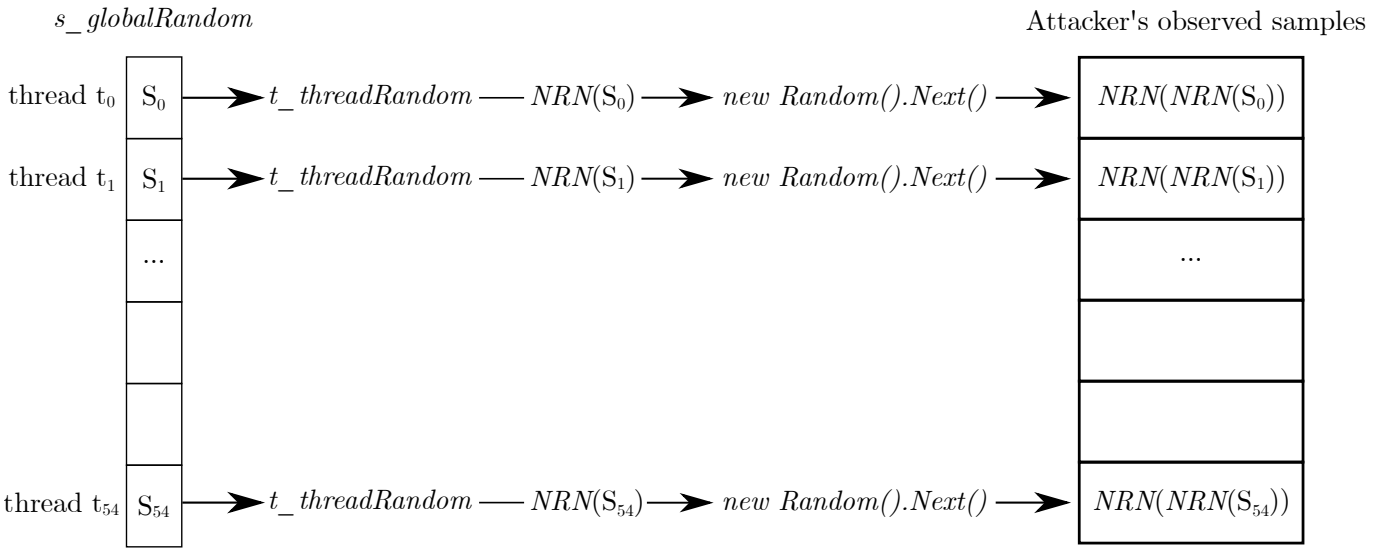

Figure 10: In the **always-new** case, the attacker observes $NRN(NRN(S_i))$

Lemma 5.
$NRN(NRN(S_i)) = NRN(NRN(S_{i-55})) - NRN(NRN(S_{i-24})) + a \cdot b + b \mod m$

$$\begin{aligned}
&\textit{With constants } a, b, m, k: \\
&a = 1121899819 \\
&b = 1559595546 \\
&m = 2^{32} - 1 \\
&k = sgn(s_b) = sgn(s_t)
\end{aligned}$$

Proof.

$$\begin{aligned}
& NRN(NRN(S_{i-55})) - NRN(NRN(S_{i-24})) + a \cdot b + b \mod m \\
&= a \cdot NRN(S_{i-55}) + b - (a \cdot NRN(S_{i-24}) + b) + a \cdot b + b \mod m && \text{[Lemma 1]} \\
&= a \cdot NRN(S_{i-55}) + b - a \cdot NRN(S_{i-24}) - b + a \cdot b + b \mod m \\
&= a \cdot (NRN(S_{i-55}) - NRN(S_{i-24}) + b) + b \mod m \\
&= a \cdot (NRN(S_i)) + b \mod m && \text{[Lemma 4]} \\
&= NRN(NRN(S_i)) && \text{[Lemma 1]}
\end{aligned}$$

□

Lemma 5 fills a similar role in this case as Lemma 4 in the single-threaded case: with its formula, the attacker can compute future output generated by the **always-new** case from past observed samples, as well as past from future.

4.3.2 Practical Hindrance: Lost Samples

The target application may be accessible not only to the attacker, but also to its regular users, who can themselves request and consume pseudorandom numbers

from the target generator. This leads to a violation of a key assumption in our described attacks: that the attacker's observed samples are **consecutive**.

This hindrance applies equally to both the **single-threaded** and the **always-new** case; and also to the **common** use case attack as in 3.2: the prediction methods we prescribe for all three of these proceed in a similar manner, and make the same assumption that 55 consecutive samples were gathered. Figure 11 illustrates one such **lost sample**.

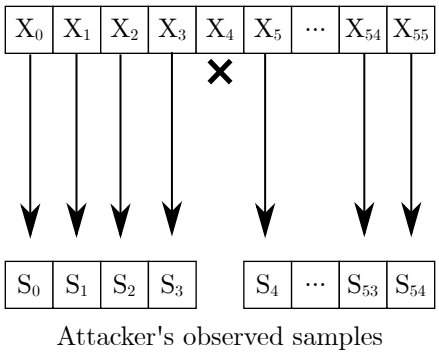

Figure 11: A user consumes X_4, breaking the attacker's samples

There is some recourse available to the attacker. **First**, lost samples can be easily detected by gathering 56 samples instead of 55: if they are not consecutive, then the 56th will not conform to the attacker's prediction as calculated from the first 55. The attacker could then *continuously* grab and validate samples from the target generator until by luck 56 of them happen to be consecutive. **Second**, the attacker may effect his attack when other users are least likely to be present; e.g. at 3 AM.

4.4 Conclusion

We introduce *NRN*(*seed*), a formula for *new Random(seed).Next()*, discovered experimentally and demonstrated by brute force. We introduce prediction methods for the target case by applying properties of *NRN* in conjunction with the seed selection methods employed by the various .NET environments — that is, a system timer in .NET Framework, and the static instances s_globalRandom and t_threadRandom in .NET Core and others. This formula and these prediction algorithms are a novelty, original to our work.

5 Tokage

In this section, we describe **tokage**, a tool implementing the **target use case** prediction methods described in the previous section: specifically, **.NET Framework** prediction, and both the **single-threaded** and **always-new** cases for **.NET Core** and others. As an extra, we also implemented **common use case** prediction as described in 3.2. **tokage**'s source code is publicly available on github [27].

5.1 Usage and Operation

tokage is a command line interface (CLI) tool written in **python 3.8.11**. Installation consists in simply copying the repository, and the tool may then be run with `python tokage.py`. No dependencies are required to run the tool other than a base python install. Figure 12 illustrates tokage's help message, with a brief description for all its arguments.

```
usage: tokage.py [-h] [-a AMOUNT] [-q TARGET_REQTIME] [-r TARGET_RESTIME] [-l L] infile target

a tool for cracking C# System.Random number generation

positional arguments:
  infile                path to input file with sampled numbers. contents must be in the following format:
                        [number1] [req1_timestamp] [res1_timestamp]
                        [number2] [req2_timestamp] [res2_timestamp]
                        ...
                        (timestamps are only needed for .NET Framework prediction)
  target                target environment/use case. avaialable options:
                        c: target the common use case. works against any .NET environment. requires at least [55] samples in infile
                        s: target single-threaded generation on .NET Core or .NET 5+. requires at least [55] samples in infile
                        n: target always-new-threaded generation on .NET Core or .NET 5+. requires at least [55] samples in infile
                        f: target .NET Framework. requires at least [2] samples in infile. also requires -q, -r, and timestamps in infile

optional arguments:
  -h, --help            show this help message and exit
  -a AMOUNT, --amount AMOUNT
                        amount of guesses to output. default is 1000
  -q TARGET_REQTIME, --target_reqtime TARGET_REQTIME
                        target sample request time, used only in .NET Framework prediction
  -r TARGET_RESTIME, --target_restime TARGET_RESTIME
                        target sample response time, used only in .NET Framework prediction
  -l L, --leniency L    maximum clock error leniency (ms), used only in .NET Framework prediction. default is 50
```

Figure 12: **tokage**'s help message, printed when the **`-h`** argument is specified

tokage is intended to receive as arguments the path to an input file containing a list of samples obtained by the target generator, and a single character specifying which prediction method should be employed. The selected method is then applied to generate and print predictions. We use the terms **guess** and **prediction** interchangeably, referring to this output. The amount of guesses produced can be set by an optional argument **`-a`**, by default 1000.

The tool operates in two different modes: one for **.NET Framework** target case prediction, and another for all three other prediction scenarios, which from now we refer to as **regular** prediction. This is because, as described in 4.3.1, these three follow similar algorithms, whereas .NET Framework prediction follows a completely different approach.

Regular prediction: the input file is expected to have *at least* **55** samples. These are then used to generate predictions by any one of Equation (2), Lemma 4, or Lemma 5; depending on which prediction method was specified. **tokage** then

outputs predictions such that the n-th prediction is expected to be equal to the n-th next sample output by the target generator.

.NET Framework target case prediction: the input file must have **2** samples, again one per line. Each sample must consist in the observed *number* along with its respective request and response times, representing when it was generated. Two additional arguments must be specified: **`-q`** for the *target sample's* request time, and **`-r`** for its response time. All timing information must have been measured with the same clock, in milliseconds. An additional optional argument **`-l`** specifies what was in 4.2.1 the maximum clock error value e, with a default of 50. The output for this mode of operation has different meaning: the n-th guess is the n-th likeliest value for the target sample.

Thus, a user is expected to first obtain such samples (and timestamps in the **.NET Framework** case), and then input these into **tokage** to effect a prediction.

In the **regular** cases, the samples are not validated; that is, no checks are performed to verify that the samples are consecutive, or even that they came from the specified use case. On the other hand, validation is indeed performed for the **.NET Framework** target case and an error is raised if the input samples are inconsistent with the target case's expected behaviour.

5.2 Implementation

Again, **tokage** is written in **python 3.8.11**. Its source code is structured into the following files and directories:

```
tokage/
├── tokage.py
├── test.py
└── modules/
    ├── __init__.py
    ├── args.py
    ├── framework/
    │   ├── __init__.py
    │   ├── fileparse.py
    │   ├── estimator.py
    │   └── timedsample.py
    └── regular/
        ├── __init__.py
        ├── fileparse.py
        ├── common/
        │   ├── __init__.py
        │   └── estimator.py
        └── core/
            ├── __init__.py
            └── estimator.py
```

- `tokage.py` is the entry-point script, i.e. the "main". It reads the command-line arguments, then reads the input samples using the appropriate `fileparse.py`, and forwards these to the appropriate `estimator.py`;
- `test.py` implements automated tests for **tokage**. It is further described in 5.5;
- `__init__.py` files are an empty files required by python to define a module or submodule folder;
- `args.py` is responsible for defining and parsing command-line arguments. It uses **argparse** from the python standard library;
- `timedsample.py` defines a class grouping a sample value with its request and response times, used for **.NET Framework target** case prediction;
- `fileparse.py` files are responsible for reading samples from a file. One such file covers all **regular** cases, and another covers **.NET Framework** target case prediction;
- `estimator.py` files finally implement prediction.

Notably, regarding the estimator files:

- `modules/framework/estimator.py` exposes a single method which implements prediction as specified in 4.2.1. An attempt is made to compensate for lag, as specified in 4.2.2. The value for $k = sgn(seed)$ for Lemma 2 is derived from the first two samples in the input file;

- `modules/regular/core/estimator.py` exposes two methods targeting **.NET Core** and other environments: one for the **single-threaded** target case, and another for the **always-new** target case. These proceed as described respectively in 4.3.1.1 and 4.3.1.2;

- `modules/regular/common/estimator.py` exposes a single method, implementing **common case** prediction as described in 3.2.

5.3 Objectives

For all three **regular** cases, **tokage**'s objective is to predict with **perfect** accuracy all future output from the target generator, **if** the input samples are consecutive. This is in fact mathematically guaranteed for our prediction methods for these three cases — see Section 3.2, Lemma 4, and Lemma 5; therefore, our tests for these cases are useful only to ensure our algorithms were implemented correctly.

For the **.NET Framework** target case, perfect accuracy is impossible — first because of *Environment.TickCount*'s low resolution, and second because of lag. These are both unpredictable phenomena. We aim to nevertheless compensate for these factors as best we can, by employing our methods as described in 4.2.2.

5.4 Evaluation Metrics

Our fundamental unit in evaluating tokage's performance is a **test run**. A test run consists in obtaining $i + m$ samples produced by the target generator, and running tokage with the first i samples as input and the last m samples as its prediction target. Recall that tokage outputs multiple guesses each time it is ran — we use g_n to denote the n-th output guess.

For all **regular** cases, we test tokage's performance to verify that it matches our theoretical expectations; that is, given 55 consecutive samples ($i = 55$), to accurately predict all $i + n$-th samples with each g_n.

For the **.NET Framework** target case, as we scope a *single* target sample at a time, it follows that $m = 1$, and the target sample is s_i. Each guess g_n is a candidate value for the target sample, with lower values of n i.e. earlier guesses being more *likely* to be the correct value as per our prediction.

For this target case, we say that a test run has a **guess score** G if $g_G = s_i$ is the correct guess for the target sample's value, and we call a test run k-successful if $G \leq k$. We suggest two evaluation metrics based on the guess score, to be computed on $T = G_0, G_1, ..., G_n$, the results of a batch with n test runs:

- $\boldsymbol{\mu_G}$, the mean value for the guess score;

- $\boldsymbol{S_k(T)/n}$, the *k-success ratio*, where $S_k(T)$ counts how many of the n test results in T were *k-successful*.

Intuitively, μ_G provides insight into tokage's general accuracy — it counts how many guesses are needed, on average, before the correct value for the target sample is found. The *k-success ratio* represents how likely it is for the correct guess to be within the first k guesses.

Furthermore, adverse network conditions will detriment **tokage**'s performance. Thus, we make an effort to run tests under various simulated scenarios. Finally, to gain statistically significant data, we run a single test run configuration multiple times, in what we call **test batches**.

5.5 Automated Tests

To gather the required data and measure our suggested metrics, we prepared and ran automated tests. These were also written in **python** and are included with tokage, in the file `test.py`. It is structured so that all relevant configuration is included at the *beginning* of the file. The tests make use of the **requests** library, which might **not** be included in a base python install.

To serve as a target for tokage's prediction attacks, we wrote a mock server named **TokageVulnExample** in C#, with its source code also published on github [28]. It is configured to build two executables: one which runs the server on .NET Framework 4.8, and the other on .NET 5. The automated tests depend on this server: those who wish to run `test.py` **must** download and build the server's source code, and locate its directory in the tests' configuration.

When ran, the server listens on **http://localhost:8080**, and upon any HTTP request to `/sample`, responds with a number generated by one of the target use cases of System.Random. Multiple command line arguments, shown in Figure 13 control how which use case is employed, and parametrise **lag simulation**, where the server waits for a time before processing and responding to the request. We include more details about these functionalities later on.

```
TokageVulnExample - an example HTTP server vulnerable to tokage's prediction attack
  -h, --help                 show this message and exit
  -m, --mean=VALUE           mean network delay
  -d, --std=VALUE            network delay standard deviation
  -f, --force                force Environment.TickCount as seed
  -s, --single               use single instance instead of new Random().Next()
  -n, --newthread            use a new thread for each request (default is
                               single-thread)
```

Figure 13: TokageVulnExample's help message, printed when the **-h** argument is specified

`test.py` starts TokageVulnExample multiple times, once for each *test batch*, with various different arguments defined by each batch. Next, the test code gathers as many samples as needed from System.Random by making multiple requests to **http://localhost:8080/sample**, and these are then used in test runs for tokage to gather the evaluation metrics as described in 5.4.

TokageVulnExample furthermore includes a webpage interface as part of a system designed to emulate scenario. This is intended for manual verification of tokage's efficacy, and demonstrative or instructive purposes. More details about this webpage

scenario are included in the project's README. Neither the interface nor any of its underlying requests are in any way used by the automated tests.

5.5.1 Regular Cases

Recall the three **regular** cases - that is, the **common use case** of System.Random, and the **single-threaded** and **always-new** threading scenarios for the target use case on **.NET Core** and other environments. The test code starts a TokageVulnExample server with the proper arguments to configure its generation to the desired target case:

- for the **single-threaded** case, the **.NET 5** build is ran with no arguments;
- for the **always-new** case, the **.NET 5** build is ran with the **`-n`** argument, specifying that a new thread handles each request;
- for the **common** use case, the **.NET 5** build is ran with the **`-s`** argument, specifying the common use case of System.Random instead of the target use case.

Each test run fetches 55 samples from the server, and uses tokage to generate n predictions from these. The tests then fetch n more samples and compare these to the predictions.

Test batches for regular cases specify which of the three regular cases is targetted, how many predictions must be verified for each run (i.e. the value for n), and how many test runs must be ran in the batch.

5.5.2 .NET Framework Target Case

Tests for this case attempt to simulate **lag**, since it is relevant to prediction quality. We modelled lag by including two delays, one intended to simulate the client-to-server network transit time, and the other for server-to-client transit time. In short, for every request received by the server:

1. It first receives the request;
2. it then waits idly for a time, marking the **first** simulated delay;
3. it then processes the request, generating a sample;
4. it then waits idly again, marking the **second** simulated delay;
5. finally, it responds to the request, sending the generated sample's value.

The amount of time waited for each delay is modelled by a random variable with $Lognormal(\mu_{ln}, \sigma_{ln})$, a distribution often used to fit or model RTT values [29]. Since the argument values μ_{ln} and σ_{ln} for a log-normal do **not** represent its distribution's final mean and standard deviation values, we derive μ_{ln} and σ_{ln} from desired mean and standard deviation values (μ_R, σ_R), as per equation (3).

$$\begin{aligned} \mu_{ln} &= \log(\mu_R) - \frac{1}{2}\log((\sigma_R/\mu_R)^2 + 1) \\ \sigma_{ln} &= \sqrt{\log((\sigma_R/\mu_R)^2 + 1)} \end{aligned} \tag{3}$$

The server derives both delays from numbers generated by a static instance of System.Random itself. For the sake of simplicity, in our tests we presume that both delays follow the *same* distribution, although reality might deviate from this assumption. We expect that in such scenarios with *different* distributions, tokage's prediction quality would worsen.

Thus, whenever the server must effect a delay: it samples a log-normal distribution, acquiring a numeric value t, and then waits for t milliseconds. One detail worthy of note - if t happens to be negative, the server simply does not wait, because it is nonsensical to wait for a negative time period.

Recall from 4.2.2 that our prediction method always prioritises the midpoint between request time and response time - this means it presumes that the two delays are likely to be nearly equal. A higher σ_R makes it more likely that they instead be imbalanced - that one delay will be significantly greater than another. Thus, we expect σ_R to have a notable effect on tokage's performance. In contrast, varying μ_R does not introduce any potential imbalance, so it can be expected to have no effect on tokage's performance.

The arguments **`-m`** and **`-d`** to TokageVulnExample set the mean and standard deviations μ_R and σ_R for the delays' distribution. **.NET Framework** target case generation is achieved by running the .NET Framework 4.8 build **without** the **`-s`** argument.

In a single test run, `test.py` successively obtains 3 samples from the server, writing down the request and response times for each. All time values are obtained with python's `time.time`. Next, it runs **tokage** with the first 2 samples as inputs, and the last sample's times as its target for prediction. The value of the guess score is found by searching for the target value in tokage's output guesses - if the G-th guess is correct, the guess score is set to G.

A test batch specifies a lag distribution (μ_R, σ_R), and how many test runs should be ran using this distribution.

5.6 Configurations and Results

5.6.1 Regular Cases

We ran three test batches, one for each of the three **regular** cases. Each batch would run 100 test runs, and each test run was configured to verify 55 predicted samples — all predictions were correct. Separately, we ran **manual** tests using 55 non-consecutive samples, for which all predictions were incorrect.

Discussion. This matches the theoretical expectation for these prediction methods: perfect output if given correct input.

5.6.2 .NET Framework target case

We ran 11 test batches for the 11 lag configurations defined by equation (4), with each batch including 2000 test runs.

$$(\mu_R, \sigma_R) = (20 * n, 10 * n) \ \forall n \mid n \in (0, 1, ..., 10) \tag{4}$$

In total, these tests took approximately 8 hours to run. Despite the large amount of time, they are not computationally expensive - most of the time is spent idly waiting for the delays incurred by lag simulation. The results are presented in figures 14 and 15

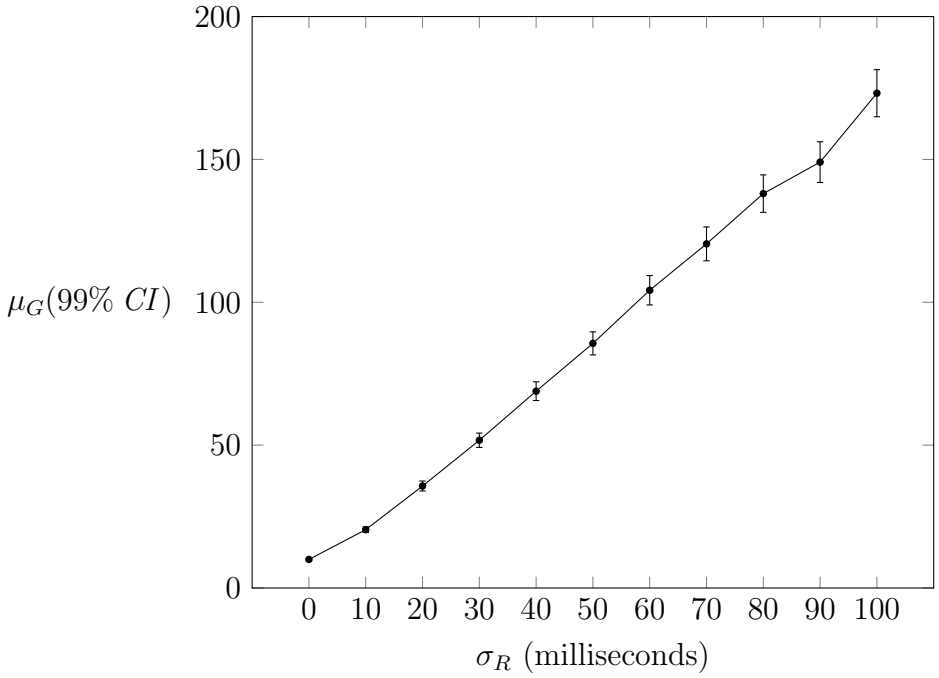

Figure 14: Mean *guess score* over various simulated network conditions.

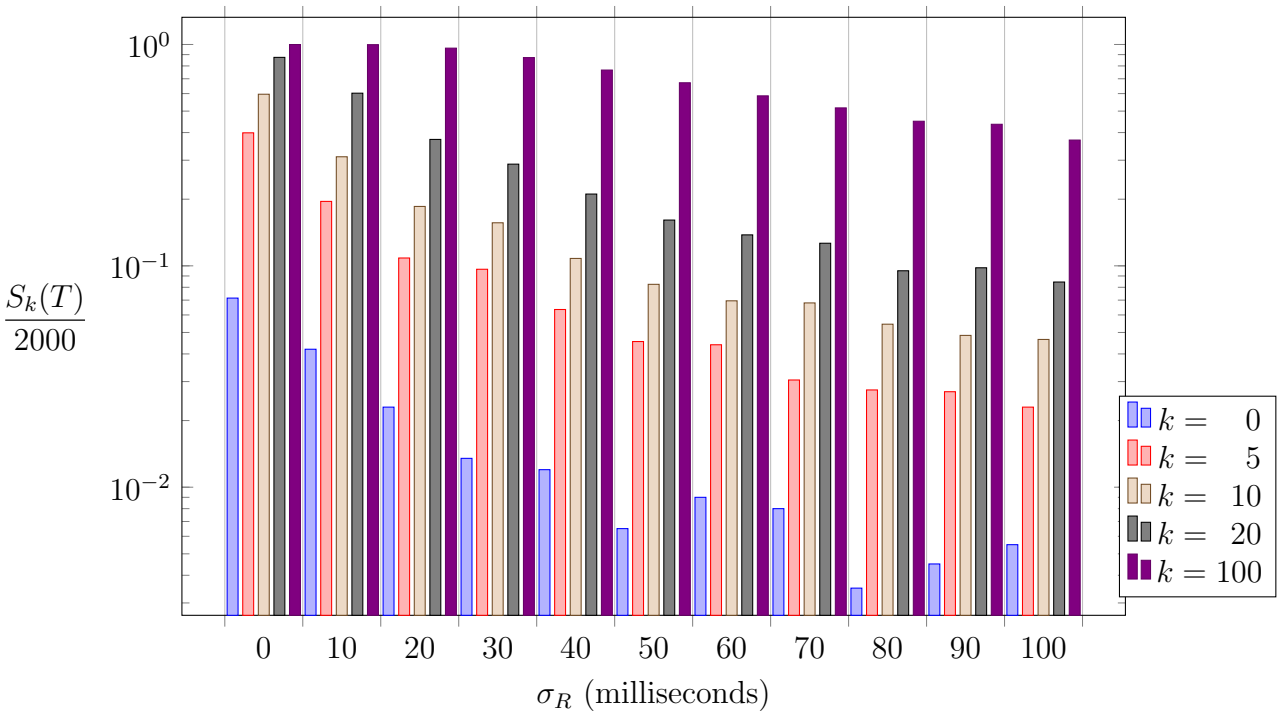

Figure 15: *k-success ratios* over various simulated network conditions.

Discussion. Guess scores increase **linearly** as network conditions worsen. The correct value is unlikely to be among the first few guesses, but chances increases as more guesses are included. Matching our theoretical arguments, tokage **never** fails to find the correct value **if** allowed to consider a large enough amount of guesses to cover all possible values for Δs as described in 4.2.2.

A tokage attack for this use case is likely to succeed over a stable connection; or even over laggy connections, if a large number of guesses may be taken into account (i.e. a large k). If neither condition can be met, the odds of success for a single attack are low, at about 2% for $k = 5$ and $\sigma_R = 100$.

5.7 Conclusion

We present our command-line System.Random predictor tool **tokage**, describe how we tested it, and show our results. For the **common** use case, as well as both **single-threaded** and **always-new** threading cases for the **target** use case on **.NET Core** and other environments: a perfect prediction is effected if correct inputs are given. For the **target** use case on **.NET Framework**, prediction accuracy is high if network conditions are favourable, or if many guesses may be taken into account.

6 Conclusion

General-purpose PRNGs are often convenient to use and quick to run, and their pseudorandom generation might even seem non-deterministic to the untrained eye. In fact, they are most often easily predictable, and thus are completely unsuitable for any context where security is involved.

In our work, we provide an overview on the principles and operation of regular PRNGs, and do the same for their security-oriented cousins, CSPRNGs. We take apart C#'s default PRNG, **System.Random**, and introduce novelty prediction methods for an unconventional yet popular use case. We package our findings into the command-line tool **tokage**, together with a sample vulnerable application **TokageVulnExample** for demonstrative purposes.

We hope that our work might in some way serve to convince both programmers and managers that the use of regular PRNGs in sensitive contexts is strictly **unacceptable**, and that these should be switched out for CSPRNGs immediately, wherever necessary.

6.1 Protection from Prediction

System.Random should **never** be used wherever predictability is unacceptable - a CSPRNG should be used in its stead. The .NET standard library offers two options which can replace System.Random wherever needed: **RNGCryptoServiceProvider** and **RandomNumberGenerator**, which are both classes defined in the System.Security.Cryptography Namespace. Neither of these is as well-documented and advertised as System.Random itself - this is perhaps a point which Microsoft could improve upon.

6.2 Future Work

First, the general theme of our work — predicting the **first** output from a freshly-initialised PRNG — could be applied to other generators. It is plausible that this may also be an unstudied yet popular recourse not only for C#, but also for other languages' default PRNG implementations. As an example, a github search for the equivalent scheme for Java's **util.Random** can return over a hundred thousand results [30].

Thread pools are a commonplace threading mechanism, with relevant consequences for the target use case in .NET Core environments. In this work, we have not found nor implemented an attack for this scenario; but we are confident such an attack must exist.

Finally, many improvements could be made to tokage to improve its ease of use or its predictive power. The potential options are too many to enumerate, but we believe the most simple of these would be to implement continuous sample validation as described in 4.3.2.

References

[1] Github, "C# search - "new Random().Next"." https://github.com/search?l=C%23&q=%22new+Random%28%29.Next%22&type=Code. *Note: you must be logged in to github in order to access the code search functionality.* Accessed: 2021-08-20.

[2] J. von Neumann, "Various techniques used in connection with random digits," in *Monte Carlo Method* (A. S. Householder, G. E. Forsythe, and H. H. Germond, eds.), vol. 12 of *National Bureau of Standards Applied Mathematics Series*, ch. 13, pp. 36–38, Washington, DC: US Government Printing Office, 1951.

[3] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[4] W. E. Thomson, "A Modified Congruence Method of Generating Pseudo-random Numbers," *The Computer Journal*, vol. 1, pp. 83–83, 01 1958.

[5] T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator," in *In ACM Transactions on Modeling and Computer Simulation*, 1998.

[6] G. Marsaglia, "The diehard random number testsuite." https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/, 1997.

[7] P. L'Ecuyer and R. Simard, "Testu01: A c library for empirical testing of random number generators," *ACM Trans. Math. Softw.*, vol. 33, Aug. 2007.

[8] J. S. Lee and G. B. Cleaver, "The cosmic microwave background radiation power spectrum as a random bit generator for symmetric- and asymmetric-key cryptography," *Heliyon*, vol. 3, no. 10, p. e00422, 2017.

[9] A. Stefanov, N. Gisin, O. Guinnard, L. Guinnard, and H. Zbinden, "Optical quantum random number generator," *Journal of Modern Optics*, vol. 47, no. 4, pp. 595–598, 2000.

[10] H. Zhun and C. Hongyi, "A truly random number generator based on thermal noise," in *ASICON 2001. 2001 4th International Conference on ASIC Proceedings (Cat. No.01TH8549)*, pp. 862–864, 2001.

[11] J. P. Mechalas, "The difference between rdrand and rdseed." Accessed: 2021-08-03.

[12] D. R. Stinson, *Cryptography: Theory and Practice*. Discrete Mathematics and Its Applications, Chapman and Hall/CRC, 3rd ed., 2005.

[13] K. Tsoi, K. Leung, and P. Leong, "Compact fpga-based true and pseudo random number generators," in *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003.*, pp. 51–61, 2003.

[14] L. Bassham, A. Rukhin, J. Soto, J. Nechvatal, M. Smid, S. Leigh, M. Levenson, M. Vangel, N. Heckert, and D. Banks, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," 2010.

[15] G. J. Mitchell and D. P. Moore, "An unpublished work." Through Knuth (1997), 1958.

[16] R. Hegadi and A. P. Patil, "A statistical analysis on in-built pseudo random number generators using nist test suite," in *2020 5th International Conference on Computing, Communication and Security (ICCCS)*, pp. 1–6, 2020.

[17] A. S. Tikhomirov, "On the program implementation of a markov homogeneous monotonous random search algorithm of an extremum," *IOP Conference Series: Materials Science and Engineering*, vol. 441, p. 012055, nov 2018.

[18] A. Muthanna, P. Masek, J. Hosek, R. Fujdiak, O. Hussein, A. Paramonov, and A. Koucheryavy, "Analytical evaluation of d2d connectivity potential in 5g wireless systems," in *Lecture Notes in Computer Science*, pp. 395–403, Springer International Publishing, 2016.

[19] R. M. Elbasiony, E. A. Sallam, T. E. Eltobely, and M. M. Fahmy, "A hybrid network intrusion detection framework based on random forests and weighted k-means," *Ain Shams Engineering Journal*, vol. 4, no. 4, pp. 753–762, 2013.

[20] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C (2nd Ed.): The Art of Scientific Computing*. USA: Cambridge University Press, 1992.

[21] R. Coveyou, "Random number generation is too important to be left to chance," *Applied Probability and Monte Carlo Methods and modern aspects of dynamics. Studies in applied mathematics*, vol. 3, pp. 70–111, 1969.

[22] H. Krawczyk, "How to predict congruential generators," *Journal of Algorithms*, vol. 13, no. 4, pp. 527–545, 1992.

[23] J. Boyar, "Inferring sequences produced by pseudo-random number generators," *J. ACM*, vol. 36, p. 129–141, Jan. 1989.

[24] Microsoft, "random.cs - random class source code for .net core." `https://archive.is/zyppT`. Accessed: 2021-07-19.

[25] Microsoft, "random.cs - random class source code for .net framework." `https://archive.is/OCDUG`. Accessed: 2021-07-19.

[26] Oracle, "Random.java - Java's random class source code." `https://archive.is/DSyfL`. Accessed: 2021-07-27.

[27] L. M. Lin, "tokage (github)." `https://github.com/cubsphere/tokage`, 2021.

[28] L. M. Lin, "TokageVulnExample (github)." https://github.com/cubsphere/TokageVulnExample, 2021.

[29] N. Brownlee and I. Ziedins, "Response time distributions for global name servers," 08 2002.

[30] Github, "Java Search - "new Random().Next"." https://github.com/search?l=Java&q=%22new+Random%28%29.Next%22&type=Code. *Note: you must be logged in to github in order to access the code search functionality.* Accessed: 2021-08-20.

A Constant Values Table for Various X_n

With $X_n = a \cdot |s| + b \mod (2^{32} - 1)$:

n	a	b
55	1121899819	1559595546
56	630111683	1755192844
57	1501065279	1649316166
58	458365203	1198642031
59	969558243	442452829
60	1876681249	1200195957
61	962194431	1945678308
62	1077359051	949569752
63	265679591	2099272109
64	791886952	587775847
65	1582116761	626863973
66	1676571504	1003550677
67	1476289907	1358625013
68	1117239683	1008269081
69	1503178135	2109153755
70	1341148412	65212616
71	902714229	1851925803
72	1331438416	2137491580
73	58133212	1454235444
74	831516153	675580731
75	285337308	1754296375
76	526856546	1821177336
77	362935496	2130093701
78	750214563	70062080
79	210465667	1503113964
80	1381224997	1130186590

81	1846331200	2005789796
82	1330597961	1476653312
83	593162892	1174277203
84	1729496551	174182291
85	792803163	401846963
86	565661843	973512717
87	863554642	638171722
88	53838754	2122881600
89	749855384	1380182313
90	93067682	1638451829
91	1778866589	65271247
92	1463507567	818200948
93	367760674	736891500
94	1219347826	2056119311
95	1648614489	1084756724
96	596622148	1537539262
97	1228675679	255459778
98	243017841	587232589
99	1132230640	1947978014
100	1891159862	1706746116
101	730619752	724046315
102	33642253	981848395
103	209795643	315304373
104	283831563	475269784
105	249493290	880625662
106	967871855	1543454120
107	1560699908	1331075398
108	437500212	1047903413
109	429989927	418573418

B NRN experimental evidence

```
using System;

namespace NRNVerification
{
    class Program
    {
        void Main()
        {
            bool allTrue = true;
            long s = int.MinValue + 1; //using long instead
                of int
            //to prevent any complications due to overflow
            for (; s <= int.MaxValue; ++s)
            {
                bool nrnCorrect = NRN(s) == new Random(s).
                   Next();
                allTrue = nrnCorrect;
                if (!allTrue) break;
            }
            if (allTrue)
                Console.WriteLine("NRN was correct for all
                   integers in range");
            else
                Console.WriteLine($"NRN incorrect for {s}")
                   ;
        }

        int NRN(int s)
        {
            // again, using long instead of int to prevent
               overflow
            long s_long = (long) s;
            return (Math.Abs(s_long) * (1121899819) +
            1559595546) % int.MaxValue;
        }
    }
}
```