Machine Learning needs Better Randomness Standards: *Randomised Smoothing* and PRNG-based attacks

Pranav Dahiya, *University of Cambridge*; Ilia Shumailov, *University of Oxford*;
Ross Anderson, *University of Cambridge & University of Edinburgh*

https://www.usenix.org/conference/usenixsecurity24/presentation/dahiya

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.
August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.
Machine Learning needs Better Randomness Standards: 
*Randomised Smoothing* and PRNG-based attacks

Pranav Dahiya  
*University of Cambridge*

Ilia Shumailov  
*University of Oxford*

Ross Anderson  
*University of Cambridge* & *University of Edinburgh*

**Abstract**

Randomness supports many critical functions in the field of 
machine learning (ML) including optimisation, data selection, 
privacy, and security. ML systems outsource the task of generating or harvesting randomness to the compiler, the cloud service provider or elsewhere in the toolchain. Yet there is a long history of attackers exploiting poor randomness, or even creating it – as when the NSA put backdoors in random number generators to break cryptography. In this paper we consider whether attackers can compromise an ML system using only the randomness on which they commonly rely. We focus our effort on *Randomised Smoothing*, a popular approach to train certifiably robust models, and to certify specific input datapoints of an arbitrary model. We choose *Randomised Smoothing* since it is used for both security and safety – to counteract adversarial examples and quantify uncertainty respectively. Under the hood, it relies on sampling Gaussian noise to explore the volume around a data point to certify that a model is not vulnerable to adversarial examples. We demonstrate an entirely novel attack, where an attacker backdoors the supplied randomness to falsely certify either an overestimate or an underestimate of robustness for up to 81 times. We demonstrate that such attacks are possible, that they require very small changes to randomness to succeed, and that they are hard to detect. As an example, we hide an attack in the random number generator and show that the randomness tests suggested by NIST fail to detect it. We advocate updating the NIST guidelines on random number testing to make them more appropriate for safety-critical and security-critical machine-learning applications.

1 Introduction

Randomness is crucial in machine learning (ML), serving a number of purposes across different areas. One use case is in federated learning, where it helps user selection to ensure a diverse representation of participants and prevent bias [34]; it can also provide privacy amplification in the process [3]. It is heavily used in optimisation, providing the foundation for Stochastic Gradient Descent [54], as well as generally for data sampling, enabling the selection of representative subsets for model training [45]. It enables Monte Carlo methods [46]. It is essential in active learning, a process where an algorithm actively selects the most informative samples for labelling, in order to reduce the training cost [31]. It forms the basis of differential privacy, the de facto standard for quantifying privacy, where noise is added to data in such a way as to protect individual privacy while still allowing for accurate analysis [19]. It contributes to the generation of synthetic data, which aids in expanding the training set and enhancing the generalisation capabilities of ML models [36]. In short, randomness is widely used and highly significant for ML.

Yet little thought has been given to the vulnerabilities that might result from weak randomness. We therefore explore the extent to which an attacker can change the safety-critical decision-making of a target system, purely by exploiting or tinkering with random number generators. We focus our efforts on *Randomised Smoothing*, a standard technique for quantifying uncertainty in a given blackbox model prediction [12]. This is heavily used in practice to combat adversarial examples; it can even be used to provide robustness certification. It samples isotropic Gaussian noise and adds it to a critical datapoint in order to measure the probability that the addition of noise causes the model prediction to change, leading to a spurious decision. Sampling high-quality noise is crucial for this purpose, yet in practice we rarely check how normal our Gaussian noise is.

In this paper, we construct two attacks against *Randomised Smoothing* that assume an attacker can influence the random number generator on which the model relies. The first is a naive attack that simply replaces a Gaussian distribution with a different distribution *e.g.* Laplace noise. This disrupts confidence quantification significantly for both over- or under-estimation, but is relatively easy to detect. We present the

---

3The literature does not view *Randomised Smoothing* as an uncertainty estimation technique, yet it is one when it is used to certify a prediction around a target datapoint.
attack intuition visually in Figure 1. Following this naive proof of concept, we construct a more powerful and covert attack: a bit-flipping PRNG attacker that changes only a single bit out of every 64 bits deep in the implementation of the random number generator. We show that this can cause mis-quantification of the true confidence by up to a factor of $\times 81$ and is extremely hard to detect. This change to the PRNG is covert, in that it does not cause it to fail the official NIST suite of randomness tests.

This highlights the inadequacy of current standards and defences in protecting against attacks targeting randomness in machine learning. We argue that similar randomness-based attacks can be devised against other ML techniques, such as differential privacy. It follows that the standards are insufficient to guarantee the security and privacy of machine learning systems in the face of sophisticated adversaries. We then discuss practical ways of tackling these vulnerabilities. We aim to empower researchers and practitioners to understand the extent to which they place their trust in their toolchain’s source of randomness, and develop more robust defences against the abuse of this trust.

To exploit a function $f : X \rightarrow \mathbb{R}$, an attacker aims to minimise $L : X \rightarrow \mathbb{R}$, where $L$ is the model’s discriminant function [6]:

$$x^* = \arg\min_{x : d(x, x^0) \leq d_{max}} L(x). \tag{1}$$

The choice of the distance function $d : X \times X \rightarrow \mathbb{R}$ is domain-specific, and the maximum size of the perturbation is limited to $d_{max}$. More efficient attacks now exist [9, 13].

2.2 Certification in Machine Learning

Certification techniques attempt to invert Equation (1) to determine the maximum perturbation radius around an input, inside which no adversarial examples exist. This is different from what the security community means by certification in the context of formal verification [15]: that refers to programmatic proof that a piece of software performs exactly as intended, matching some high-level abstraction. For example, a certified compiler will always correctly translate legal code correctly into its compiled binary or throw an appropriate error [39]. If the high-level abstraction of a machine learning model is considered to be the function $f : X \rightarrow \mathcal{Y}$, mapping inputs to their true classes, then certification here would imply that, for a given volume around a point from $X$, the prediction does not change from $\mathcal{Y}$. Many certification techniques in current use are approximate and, despite the fact that theorems may be offered around their behaviour, they only provide probabilistic guarantees for the absence of adversarial examples in an $\varepsilon$ ball around a given input [13].

Randomised Smoothing is not the only certification technique. Other mechanisms exist, e.g., by constructing a bounding polytope instead of a hypersphere, or using either linear relaxation or interval bound propagation [67]. However,
these require changes to the training process and model re-engineering, which in turn makes them hard to scale up to more complex model architectures and higher dimensional data [15]. Randomised Smoothing is one of the most widely used certification techniques and it underpins current state-of-the-art certification algorithm for machine learning models [68]. It can be applied to any underlying model in a black-box manner and scaled up to larger model architectures. It is used both for security, i.e. to combat adversarial examples, and safety, i.e. to provide confidence in predictions. That is why we made it the target of this work.

2.3 Randomised Smoothing

Any classifier \( f \) can be used to construct a smoothed classifier \( g \) such that:

\[
g(x) = \arg \max_{c \in \mathcal{Y}'} \mathbb{P}\{ f(x + \epsilon) = c \}, \quad \text{where} \quad \epsilon \sim \mathcal{N}(0, \sigma^2 I).
\]  

(2)

This formulation allows for easy approximation of \( g \) for a given confidence bound using a Monte Carlo algorithm by sampling \( \epsilon \) from \( \mathcal{N}(0, \sigma^2 I) \) [12]. While previous work used differential privacy [18, 37] and Renyi divergence [40, 64] to determine the radius of the hypersphere around an input inside which the absence of adversarial examples can be provably verified, Cohen et al. [12] used the Neyman-Pearson lemma [48] to obtain a certification radius which is also provably maximal. The certified radius obtained from randomised smoothing is:

\[
R = \frac{\sigma}{2} \left( \Phi^{-1}(p_A) - \Phi^{-1}(p_B) \right),
\]  

(3)

where \( \Phi^{-1} \) is inverse of the standard normal cumulative distribution function, \( p_A \) is the probability of the most probable class \( c_A \in \mathcal{Y}' \) and \( p_B \) is the probability of the next most probable class. This result holds true even if \( p_A \) and \( p_B \) are replaced with \( \tilde{p}_A \) a lower bound on \( p_A \), and \( \tilde{p}_B \), an upper bound on \( p_B \) such that \( p_A \geq \tilde{p}_A \geq \tilde{p}_B \geq p_B \). The mathematical proof for Equation (3) and its maximality can be found in the appendices of the full version of the original paper by Cohen et al. [12]. Using these results, an algorithm for certifying the predicted class from an input \( x \) can be constructed as follows.

For a base classifier \( f \) and input \( x \), \( n_0 \) counts of noise are sampled from \( \mathcal{N}(0, \sigma^2 I) \), and the modal class \( c_A \) is selected as the target class. In order to perform the certification, the output classes are sampled under noise from \( f, n \) times. A lower confidence bound for \( p_A \) with confidence \( \alpha \) is obtained using the Clopper-Pearson method [11]. \( \tilde{p}_B \) is estimated as \( 1 - p_A \). Finally, the radius of certification can be computed using Equation (3). While \( n_0 \) can be relatively small to determine \( c_A \) effectively, \( n \) needs to be quite large. Approximately \( 10^5 \) samples are required to certify a radius of \( 4\sigma \) with 99.9% confidence [12].

2.4 Randomness in (Adversarial) ML

Randomness is heavily used in machine learning. Stochastic Gradient Descent, the randomised optimisation algorithm that arguably enabled modern deep learning, relies on randomness for data sampling [54]; randomised dropout improves generalisation [21]; active learning approaches rely on biased samplers to enable faster learning [31]; randomised transformations are used to reduce overfitting [40]; while randomised sampling of data can lead to improved privacy [3], and even bound privacy attacks [63].

Adversarial ML uses randomness in both attack and defense. Adversarial examples benefit significantly from random starts [13], while many ML system engineers advocate defences based on random pre-processing where inputs are stochastically transformed before inference [2, 22, 50]. We are aware of only two attacks so far that depend on exploiting randomness: the batch reordering attack [60] and the randomised augmentation attack [52], where the order of data and the randomness of the augmentation are manipulated respectively to introduce backdoors into a target model.

Instances of randomness failure are not uncommon, especially in the context of differential privacy, where they have occasionally led to safety and security issues. A noteworthy example dates back to 2012 when Mironov discovered that the textbook implementations of floating point numbers for Laplace noise sampling resulted in an inaccurate distribution of sampled values [47]. This led to a violation of privacy guarantees and highlighted the significance of the problem. More recently, a timing side-channel was discovered in the implementation of randomness samplers, once more violating privacy [28]. Finally, randomness plagues reproducibility in ML – model training is highly stochastic [69], which is in tension with repeatable model training [27].

3 Prior on Randomness

Monte Carlo algorithms like the one described in Section 2.3 require random sampling. This immediately leads an inquisitive mind to question, “what is a random number, and how can one be generated?” The mathematical definition of a random sequence of numbers has been the subject of much debate since before computer science came into existence. For an in-depth discussion of random sequences and their limitations, the reader is referred to chapter 3.5 of The Art of Computer Programming by Knuth [33]. What follows is a summary of some points relevant to this paper.

3.1 Random Number Generators

While many distributions can be sampled to generate random sequences, in the context of computer programming, generally the uniform distribution \( \mathcal{U}(a, b) \) is used, i.e. every number
between \(a\) and \(b\) has an equal probability of selection. As discussed later in Section 3.3, the standard uniform distribution (\(U(0, 1)\)) can be transformed into any other random distribution and is therefore a natural starting point for random number generators (RNG). There are quite a few sources of entropy that a computer can use to generate random numbers such as the timing of keystrokes or mouse movements [20]. For example, consider an RNG relying on timing of keystrokes as the source of entropy. Generating one random bit from this RNG can be modelled as an experiment phrased as “Is the time taken between two keystrokes in milliseconds an even number?”. The outcome of this experiment can either be 0 or 1, thereby generating a random bit. The function that assigns a real value to each outcome of a random experiment such as the one described here is called a random variable [55]. The distribution function of a 1D random variable \(X\) over the real line can be defined as

\[
F(x) = P\{X \leq x\} = P\{X \in (-\infty, x]\}. \tag{4}
\]

At the dawn of computing, a lot of research effort was spent on developing efficient ways of generating random numbers. There were mechanical devices such as ERNIE [29] which was used to generate random numbers for an investment lottery and could possibly be attached to computers. Modern CPUs now often feature built-in hardware RNGs that use miniscule natural fluctuations in current to generate random bits [62]. However, the limitations of using mechanical RNGs in the 1950s are applicable to these modern hardware RNGs as well. First, a hardware RNG makes it difficult to reproduce the functioning of a randomised program to check if it is behaving as expected. There is also a possibility that the machine will fail in ways that would be difficult to detect by a program using it [33]. Another problem is that is difficult to judge the level of entropy of an RNG relying on real-world properties [20]. Finally, of special interest to the problem being tackled in this work is the fact that it is very tricky to determine the distribution function of a hardware RNG, as this can depend on environmental factors outside the control of the system designer.

These issues led to the development of pseudo-random number generators (PRNGs) which rely on deterministic calculations to generate sequences of numbers that are not truly random, yet appear to be. Von Neuman [66] proposed the first PRNG in 1946 using the middle-square method. PRNGs use the current number in the sequence or some other counter to generate the next number, but they need a starting point – a seed – to generate the first number or counter value. The sequence of numbers is thus a function of the seed, and two instances of the same PRNG will produce the same sequence if the seeds are identical\(^2\). The following sections present the process of generating random numbers and transforming them into samples from the normal distribution. This is followed by an overview of statistical tests to determine the quality of random numbers produced by a PRNG, and finally a discussion of possible attacks.

### 3.2 The Linear Congruential Method and PCG64

Most popular random number generators use the linear congruential method, first introduced by Lehmer in 1949 [38]. This produces a sequence of numbers \(X_1, X_2, \ldots\) as follows [32]:

\[
X_{n+1} = (aX_n + c) \mod m, \quad n \geq 0, \tag{5}
\]

where:

- \(m > 0\) is the modulus,
- \(0 \leq a < m\) is the multiplier,
- \(0 \leq c < m\) is the increment, and
- \(0 \leq X_0 < m\) is the seed.

Equation (5) can be used to generate random numbers between \(0\) and \(m\). Different choices of \(a\) and \(c\) affect the performance. The permuted congruential generator (PCG) [49] is widely used and uses an adaptation of this method. The 64-bit version has a 128-bit state variable, which is advanced according to Equation (5), i.e. \(X_n\) gives the \(n\)th state and is a 128-bit integer. A 64-bit random number can be generated from this 128-bit state as:

```
output=rotate64 (state^rotate64 (state>>64)), state>>122)
```

First, the state is bit shifted right by 64 bits and XORed with itself to improve the randomness in high bits. Then a clockwise rotational bit shift of \(r\) bits is applied to the lower 64 bits of the resulting value, where the value of \(r\) (between 0 and \(2^6\)) comes from the leftmost bits of the PRNG state\(^3\). For more details on the design choices and an empirical analysis of its performance, the reader is referred to O’Neill [49]. The PCG64 generator is the default choice for the Numpy library [25] and is the target of the attack presented in this work. Other popular random number generators include the Mersenne Twister [24, 42], SFC64 (Small Fast Chaotic) [17] and Philox [57]. Pytorch uses Philox, which is a counter passed through a block cipher. As it was designed to achieve high performance in HPC applications, the cipher was deliberately weakened [57].

### 3.3 Sampling from a Normal Distribution

The numbers generated by PCG64 can be assumed to be sampled from the discrete uniform distribution \(U(0, 2^{64})\). However, most randomised algorithms need this to be transformed

\[^2\]Not all PRNGs work like this, but this is a desirable property to ensure reproducibility. Cryptographic PRNGs typically combine hardware and pseudorandom components in such a way that both have to fail to make key material easily predictable by an opponent.

\[^3\]Typecasting the 128-bit state to uint64_t in C retains the lower 64 bits and discards the top 64 bits.
into a different distribution, usually $\mathcal{U}(0, 1)$ or $\mathcal{N}(0, 1)$, which can then be transformed into any uniform or normal distribution trivially. Transforming discrete $\mathcal{U}(0, 2^{54})$ to continuous $\mathcal{U}(0, 1)$ can be done by simply discarding the first 11 bits of the random number\(^4\), typecasting as double, and dividing by $2^{53}$ to effectively shift the decimal point from after the least significant bit to before the most significant bit.

The Ziggurat method [44] is state-of-the-art when it comes to sampling from decreasing densities such as the normal distribution. At a high level, this involves dividing the distribution into smaller pieces, choosing one of these pieces randomly, and then sampling from it. The smaller pieces into which the distribution is divided are chosen such that the areas under the density curve are equal, i.e., each piece is equally likely (shown in Figure 2). The lower 8 bits (index bits) are used to select one of these pieces at random, the next bit (sign bit) is used to randomly assign a sign, and finally, the remaining bits (distribution bits) are used to sample a float from the uniform distribution corresponding to the rectangular area selected using the index bits.

3.4 Attacks, Dual_EC_DRBG and Bullrun

Traditional attacks on pseudo-random number generators have focused on determining the next number to be generated in a sequence; or more generally, given some of the numbers in a sequence, deducing some others. If the generator is used to produce cryptographic keys and also other random values that may become visible to the attacker, such a deduction may compromise an encrypted message. Such attacks aim to determine the state of the PRNG being used by the victim. Since most PRNGs are deterministic, a known state can be used to determine future numbers in the sequence [20]. Another class of attacks involves introducing a backdoor in the PRNG to make future outputs more predictable. The NSA pushed NIST to include EC-DRBG in the ANSI X9.82 standard for random number generators when the standard was first developed at the start of the 21st century. The PRNG featured in the first draft of the standard, which was published in 2004. The research community expressed concerns about EC-DRBG not being cryptographically sound by 2005 [23] – before the first official version of the standard was published [4] – leading to conjecture of an NSA backdoor [58]. Despite these concerns, EC-DRBG was adopted as the default random number generator by RSA in their BSAFE cryptography library [14]. The Snowden leaks in 2013 confirmed the existence of project Bullrun, which aimed “to covertly introduce weaknesses into the encryption standards followed by hardware and software developers worldwide” and successfully injected a backdoor into the default PRNG used for all cryptographic encryption between 2006 and 2013. This is a useful reminder of the tactics, tools, and procedures that may be used by a capable motivated adversary to carry out a backdoor attack against pseudo-random number generators.

4 Methodology

4.1 Threat Model

It is assumed that the victim is attempting to use ML to get predictions in a safety-sensitive or security-sensitive setting. Therefore it is important to accurately gauge the robustness of every prediction, and we assume that Randomised Smoothing is being employed for this purpose. By definition, the probability that an adversarial example can be found within the certified radius obtained from randomised smoothing is low: less than 0.1% if $\alpha$ is set to 0.001 as suggested by Cohen et al. [12]. The knowledge that the victim is employing randomised smoothing can in itself be very useful to an adversary. This was demonstrated by Cullen et al. [15], who used this information to only search for adversarial examples with $l^2$ distance greater than the certified radius, achieving better than the state-of-the-art success rate at finding adversarial examples against ML models using randomised smoothing. Furthermore, in their evaluation of randomised smoothing, Cohen et al. [12] found that the probability of finding adversarial examples increases rapidly as the upper bound on the $l^2$ norm of the adversarial example set by the adversary increases beyond the certified radius, $R$. For the ImageNet dataset [16], they found that the probability of finding an adversarial example against a smoothing classifier is 0.17 at an upper bound of $1.5R$ and 0.53 at $2R$. Hence, confidence serves a good proxy for reliability of prediction, and any compromise of Randomised Smoothing will decrease its trustworthiness.

The objectives of an attacker attempting the class of attacks presented in this paper are twofold, depending on whether the certified radius obtained from randomised smoothing is being manipulated to be higher or lower. A higher certified radius can make the victim believe that the prediction is more robust for a given input than it actually is, so that the victim ignores adversarial examples within the spoofed radius. And certification is costly, requiring approximately $10^5$ inferences from the smoothed model to certify a radius of $4\epsilon$. Reducing the certified radius can make the victim waste time and compute, forcing them to generate more predictions under noise.

---

\(^4\)These bits contain information about the exponent in 64-bit floating point numbers [1]
to obtain the desired radius, providing a service-denial attack.

In this paper we limit our adversary access to the underly-
ing random number generator used by the victim to perform randomised smoothing. While the setting may seem unreal-
istic, an attack with such access happened in the past. The
story of the Dual_EC_DRBG PRNG, which the NSA used to
compromise RSA libraries, was told in Section 3.4.

The setup used by the victim to perform training and certifi-
cation can be quite complicated, leaving multiple avenues for
an attacker to gain access and manipulate the noise. Modern
practices such as ML-as-a-service, and outsourcing of training
and data generation to third-parties, opened a Pandora’s box.
The attacks discussed here focus on modifying the noise dis-
tribution, first by using a different noise function directly and
later by modifying the bitstream generated by an underlying
pseudo-random number generator (PRNG). Where the victim
has little or no control over the software and hardware used for
training and certification, there are even more attack vectors.

The objective of the attacks we present in the following sec-
tions is to spoof the certified radius while escaping detection
by the victim  whether by analysing the performance of the
model or by using statistical tests (discussed in Section 4.4).
This class of attacks can be carried out in a traditional way
by exploiting the hardware or software layers. In addition,
with recent developments in denoising diffusion models [26],
they can be carried out in an ML-as-a-Service scenario too.
If the victim submits inputs to a cloud API for prediction
and certification, an attacker can effectively remove the noise
introduced by the victim and replace it.

4.2 Naive noise distribution replacement

In order to check the feasibility of the attacks described in
the previous section, an initial test was done by explicitly
changing the distribution that is sampled during certification.
The GitHub repository released by Cohen et al.3 was used as
a starting point. Cohen et al. [12] reported that the variance
of distribution that additive noise is sampled from controls
the trade-off between accuracy and robustness. A lower value
of $\sigma$ can be used to certify smaller radii but with a high degree
of accuracy, whereas a higher value of $\sigma$ is needed to certify
large radii, resulting in lower accuracy. $\sigma = 0.25$ was found
to achieve an acceptable trade-off between certification radius
and accuracy on the CIFAR 10 [35] dataset, which was used to
run all the experiments in this paper. Using a base clas-
sifier trained with additive noise sampled from $\mathcal{N}(0, 0.25^2)$,
the naive attack swaps the noise distribution for one of the
following when certification is performed: Laplace distribution,
$\mathcal{L}(0, 0.25)$; half-normal distribution, $|\mathcal{N}(0, 0.25^2)|$; uniform
distribution, $\mathcal{U}(-0.25, 0.25)$; and Bernoulli distribution,
$\mathcal{B}(0.5)$. This naive attack was used to demonstrate the feas-
bility of this class of attacks on PRNGs. The next section will
describe a more sophisticated attack.

4.3 Bit-flipping PRNG attacker

The objective of the bit-flipping PRNG attacker is to modify
the stream of bits being generated by the random number
generator to alter the distribution when normal variates are
sampled. An overview of the algorithm used to transform
random integers to floats in the standard normal distribution
is presented in Section 3.3. Taking a 64-bit random integer, the
rightmost 8 bits determine the rectangle in which the normally
distributed random number will fall. These are called the index
bits. The 9th bit is the sign bit. Finally, the remaining 55 bits
are transformed into a floating point number falling within
the limits of the uniformly distributed rectangle chosen by the
index bits. These are called the distribution bits. The following
attacks modify one of these three categories of bits to alter the
resulting distribution. Altering a distribution can be done by either
introducing kurtosis or skewness. Kurtosis is a measure of the
tailedness whereas skewness is a measure

---

3Note that this may result in quantisation artefacts, e.g. pixel with value 245 noised to 235.3, gets quantised to 236, and poten-
tially change performance of Randomised Smoothing.

---

https://github.com/Xzh0u/randomized-smoothing
of the distribution around the mean.

All attacks described in this section were performed on the PCG64 random number generator [49] in the numpy library [25], which was then used to sample noise when performing randomised smoothing. PCG64 is a NIST-certified pseudo-random number generator. For an overview of how it works, the reader is referred to Section 3. The following attack was performed by modifying the next64 function, which is used to generate a random 64-bit integer, in the pcg64.h file. The original version of the function is shown in Listing 1.

4.3.1 Negative Kurtosis Attack

The uniform distribution increased the relative certified radius the most and it builds the intuition behind the negative kurtosis attack. By modifying the distribution bits, so that they are no longer uniformly distributed, but positively skewed, the kurtosis in the normal distribution is reduced, as there are fewer random numbers near the mean and more towards the tails. The modification to the PCG64 code is shown in Listing 2. Consider a uniformly distributed random variable, $X \sim U(0, 1)$. The probability density function (pdf) for $X$ is:

$$p(x) = \begin{cases} 1 & 0 < x < 1 \\ 0 & \text{otherwise} \end{cases}. \quad (6)$$

This can be skewed to a new random variable $X'$ by altering the pdf as follows (for $b$ in Equation (11)):

$$p'(x) = \begin{cases} ax + b & 0 < x < 1 \\ 0 & \text{otherwise} \end{cases}. \quad (7)$$

In order to transform $X$ to $X'$, first, the cumulative distribution function, $F'$ of $X'$ must be derived.

$$F'(x) = \int_0^x p'(y) \, dy = \int_0^x ay + b \, dy$$

$$= \left[ \frac{ay^2}{2} + by \right]_0^x = \frac{ax^2}{2} + bx. \quad (9)$$

Since $F'(x) : (0, 1) \to (0, 1)$, and $0 < X < 1$, $X$ can be transformed to $X'$ by inverting $F'$. For $x \in X$, the corresponding $x' \in X'$ can be computed as follows:

$$x' = F'(x) = \frac{-b + \sqrt{b^2 - 2ax}}{a}. \quad (10)$$

The degree of skewness in $X'$ can be controlled by changing $a$. Since $F(1) = 1$ from the definition of $X'$,

$$b = 1 - \frac{a}{2}. \quad (11)$$

In Listing 2, the tunable parameter is $\alpha$, such that $a = 1/\alpha$. First, the 64-bit random integer is converted to a 64-bit double-precision floating point number. Then, the transformation from $X$ to $X'$ is applied. Next, the number is converted back to a 64-bit integer. This is finally bit-shifted left by 9 bits and the least significant 9 bits are copied over from the original random integer generated by the PRNG. This is so that the index and sign bits remain random, and only the distribution bits are modified. A lower value of $\alpha$ results in higher skewness in the distribution bits, and more negative kurtosis in the resulting distribution. The sampled probability densities are plotted in Figure 3a, along with a reference curve of the probability density function of the normal distribution.

4.3.2 Skewness Attack

The half-normal distribution achieved a significant success rate in manipulating the certified radius to be high enough such that evasion attacks could succeed, and yet still maintain an acceptable level of performance compared to the original model. Therefore, this distribution formed the basis of the skewness attack shown in Listing 3. An 8-bit integer is used as a counter, initialised to 1. Every time a random integer with the sign bit set to 1 is encountered, the counter is shifted left. When the value of the counter reaches the tunable parameter, $\beta$, the sign bit is flipped to zero, keeping the rest of the random number the same. This results in the final normally distributed random number being positive instead of negative, thereby skewing the distribution. The sampled probability distributions for $\beta = 0, 1, 2$, and 4 are shown in Figure 3b. For $\beta = 0$, the resulting distribution is the same as $\beta \mathcal{N}(0, 0.25^2)$.

4.3.3 Positive Kurtosis Attack

The motivation for this attack comes from the increase in certified radius observed when the normal distribution was replaced with the Laplace distribution. While the relative increase in radius achieved was not as high as the others, the attack is further motivated by the reasoning that it is more likely that the sampled prediction is the same as the modal class if the $l^2$ norm of the additive noise is lower, thereby increasing the certified radius. The modified function in PCG for this attack is shown in Listing 4.

A counter is initialised to 1 and shifted left every time a random number is generated. Every time the value of the counter is equal to the tunable parameter $\gamma$, the index bits in the random number are shifted right by 1. This divides the value of the index and reduces the size of the rectangular region chosen by the algorithm used for transformation to the normal distribution. It moves the random normal numbers closer to the mean, and increases the kurtosis of the resulting distribution, with the maximum kurtosis achieved for $\gamma = 0$, and becoming closer to the standard normal distribution as $\gamma$ increases. The sampled probability densities of the distributions resulting from $\gamma = 0, 1, 2$ and 4 are shown in Figure 3c.
\( \alpha = 1 \) \hspace{2cm} \alpha = 2 \hspace{2cm} \beta = 0 \hspace{2cm} \beta = 1 \hspace{2cm} \gamma = 0 \hspace{2cm} \gamma = 1 \hspace{2cm} \alpha = 3 \hspace{2cm} \alpha = 4 \hspace{2cm} \beta = 2 \hspace{2cm} \beta = 4 \hspace{2cm} \gamma = 2 \hspace{2cm} \gamma = 4

(a) Negative kurtosis attack \hspace{2cm} (b) Skewness attack \hspace{2cm} (c) Positive Kurtosis attack

Figure 3: These figures show the sampled probability densities against the normal pdf in orange for all considered values of the tunable parameters for each of the bit-flipping PRNG attacks proposed. The negative-kurtosis, skewness and positive-kurtosis attacks all alter the sampled distribution in a self-explanatory manner.

4.4 Defences

4.4.1 NIST Test Suite

The National Institute of Standards and Technology (NIST) first published the SP800-22 test suite in 2001, in the aftermath of the competition in which the AES algorithm was selected. Its purpose was to ensure that random number generators used for cryptography are secure. This is done via a set of 15 statistical tests that operate on bitstreams from a random number generator, aiming to detect any deviation from a truly random sequence of bits. The reader is directed to Bassham et al. for an in-depth description of each test along with the recommended parameter values to use [5]. It is important to note that this test suite is a measure of PRNG badness and not goodness. According to the recommendations presented by NIST, it is important that a PRNG’s design is based on sound mathematical principles. Due to the statistical nature of these tests, it is possible that a well-designed PRNG may sometimes fail a test, while a bad PRNG may pass some tests, as demonstrated later in this paper.

4.4.2 Test for Normality

We employ a number of statistical tests to detect finer fluctuations in the resulting distribution after transformation and to quantify the deviation. In the main body of the paper, we use the Shapiro-Wilk test [59] – empirically one of the highest-power statistical tests to determine whether a set of samples is normally distributed [53]. We list the performance of other tests in Appendix D. The test statistic for this test is defined as:

\[
W = \frac{\left(\sum_{i=1}^{n} a_i x_i\right)^2}{\sum_{i=1}^{n} (x_i - \bar{x})^2},
\]

where:

- \( x_i \) is the \( i \)th order statistic,
- \( \bar{x} \) is the sample mean,
- \( m = (m_1, \ldots, m_n)^T \) are the expected values of the order statistics,
- \( a = m^T V^{-1} \), for
- \( V \) the covariance matrix of \( m \),
- \( C = \|V^{-1}m\| \).

The distribution of \( W \) does not have a name, and the cutoff values for it are computed using Monte-Carlo simulations. \( W \) lies between zero and one – the closer it is to one, the more likely it is that the samples belong to the normal distribution. The \( p \)-value determines the confidence in the test statistic.

5 Evaluation

5.1 Experimental Setting

Setting: Certification is performed on a subsample of 500 images from the CIFAR 10 test set [35], each with 100 noise samples for selection \((n_0)\), 10,000 noise samples for certification \((n)\), and \( \alpha = 0.001 \). We use the same base model as Cohen et al. [12], with a ResNet-110 architecture.

Measurements: Two types of results are collected: the first is the accuracy of the smoothed classifier, and the second is the size of the certification radius. The accuracy results attempt to determine if a naive defender will be able to detect the attack by simply observing the performance of the model. These are presented as a relative confusion matrix compared...
to the baseline unperturbed classifier, presenting the number of predictions that are correct, incorrect or abstains. The results for the impact of each attack on the certified radius were collected for the images where both the attacked model and the baseline predicted the same class. The fraction of images for which the relative certified radius \( (R' / R) \), where \( R' \) is the manipulated radius and \( R \) is the original radius) falls in each of the bins: \((0, 1.0]\), \((1, 1.1]\), \((1.1, 1.25]\), \((1.25, 1.5]\), \((1.5, 2.0]\) and \((2.0, \infty)\), is reported in Table 2. The higher the value of \( R' / R \), the easier it will be to find adversarial examples within the manipulated radius \( R' \). According to Cohen et al. [12], there is a 17% probability of finding an adversarial example with \( d_{\text{max}} = 1.5R \) and 53% probability for \( d_{\text{max}} = 2R \), where \( d_{\text{max}} \) is as defined in Equation (1).

**Defences:** More sophisticated defences going beyond observing the change in performance of the classifier were also evaluated and are considered in Section 5.4. First, the NIST test suite for certifying cryptographically secure PRNGs was run for various lengths of bit streams, ranging from \( 10^3 \) to \( 10^6 \), with 1000 tests in each run. The official version of the NIST test suite was used. For each PRNG tested, raw 64-bit integers are generated, and the binary representation is saved as ASCII-encoded strings in a text file. This is used as an input for the test utility. The next defence evaluated is the Shapiro-Wilk test for normality. First, raw 64-bit integers were transformed to the normal distribution using the default implementation in numpy [25], and then tested using the AS R94 algorithm [56] from the scipy library [65]. This algorithm is limited to a maximum of 5000 samples, which is therefore the number of samples used to run our test.

| Table 1: The above tables show a relative confusion matrix of the performance of Randomised Smoothing subject to naive attacks, where the normal distribution is swapped for a different distribution. The uniform distribution leads to the greatest deviation from baseline performance, followed by the bernoulli distribution, then the absolute normal distribution, and finally the laplace distribution.

| Distribution               | \( L(0, 0.25) \) | \( |X(0,0.25^2)| \) | \( U(-0.25,0.25) \) | \( B(0.5) \) |
|---------------------------|-----------------|-----------------|-----------------|----------------|
| Correct                   | 282             | 9               | 128             | 154            |
| Incorrect                 | 48              | 52              | 70              | 42             |
| Abstain                   | 44              | 19              | 11              | 107            |
| Absolute normal distribution               |                  |                  |                  |                |
| Correct                   | 287             | 73              | 235             | 132            |
| Incorrect                 | 12              | 62              | 1               | 5              |
| Abstain                   | 15              | 24              | 7               | 4              |
| Laplace distribution                       |                  |                  |                  |                |
| Correct                   | 25               | 9               | 48              | 24             |
| Incorrect                 | 52              | 5               | 44              | 42             |
| Abstain                   | 19              | 20              | 19              | 17             |
| Bernoulli distribution                      |                  |                  |                  |                |
| Correct                   | 113             | 154             | 113             | 154            |
| Incorrect                 | 42              | 42              | 33              | 42             |
| Abstain                   | 33              | 17              | 33              | 17             |

5.2 Naive attacker

The change in classifier accuracy with the naive attack is presented in Table 1. As highlighted earlier, a smoothed model will never perform as well on the classification problem, as it becoming less accurate as more noise is introduced. Therefore, the attacker can get away with slightly reducing the accuracy and still escape detection since this is expected behaviour. \( L(0, 0.25) \) and \( |X(0,0.25^2)| \) perform well as attack noise distributions in this regard, with only nominally reducing model accuracy. The other two attack distributions \( U(-0.25,0.25) \) and \( B(0.5) \) result in significantly reduced accuracy, which will raise red flags for the defender.

Based on the discussion in Section 4.1, the probability of finding an adversarial example increases considerably as the relative manipulated radius increases. The focus here is on values of \( R' / R \) that lie in \((1.5, 2.0]\) and \((2.0, \infty)\), which give the attacker a significantly high probability of performing an evasion attack. \( |X(0,0.25^2)| \) performs really well in this regard, with 28% of input images having a spoofed radius that falls in the highly vulnerable categories. \( U(-0.25,0.25) \) achieves the best attack success rate, but it is also the easiest attack distribution for the defender to detect by simply monitoring the performance of the classifier.

5.3 Bit-flipping PRNG attacker

In this subsection we evaluate the three attacks described in Section 4.3: negative-kurtosis, skewness and positive-kurtosis. The parameter values chosen for evaluation are \( \alpha \in \{1, 2, 3, 4\} \), \( \beta \in \{0, 1, 2, 4\} \) and \( \gamma \in \{0, 1, 2, 4\} \) for each of the attacks respectively. Note that none of these attacks can be detected by a naive defender. Appendix B shows the relative accuracy results, that do not deviate significantly from the baseline. Relative certification radii for each attack are shown in Table 2, with the skewness attack achieving the
<table>
<thead>
<tr>
<th>$\frac{R'}{R}$</th>
<th>$\frac{R'}{R} &lt; 1$</th>
<th>$1.0 &lt; \frac{R'}{R} &lt; 1.1$</th>
<th>$1.1 &lt; \frac{R'}{R} &lt; 1.25$</th>
<th>$1.25 &lt; \frac{R'}{R} &lt; 1.5$</th>
<th>$1.5 &lt; \frac{R'}{R} &lt; 2.0$</th>
<th>$\frac{R'}{R} &gt; 2.0$</th>
<th>max($\frac{R'}{R}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L(0,0.25)$</td>
<td>0.80</td>
<td>0.07</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.07</td>
<td>6.03</td>
</tr>
<tr>
<td>$</td>
<td>\mathcal{N}(0,0.25^2)</td>
<td>$</td>
<td>0.24</td>
<td>0.31</td>
<td>0.09</td>
<td>0.08</td>
<td>0.10</td>
</tr>
<tr>
<td>$U(-0.25,0.25)$</td>
<td>0.32</td>
<td>0.22</td>
<td>0.03</td>
<td>0.08</td>
<td>0.07</td>
<td>0.28</td>
<td>13.99</td>
</tr>
<tr>
<td>$B(0.5)$</td>
<td>0.88</td>
<td>0.02</td>
<td>0.01</td>
<td>0.02</td>
<td>0.03</td>
<td>0.04</td>
<td>9.58</td>
</tr>
</tbody>
</table>

**Naive noise distribution replacement attack**

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$\beta$</th>
<th>$\gamma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha = 1$</td>
<td>$\beta = 0$</td>
<td>$\gamma = 0$</td>
</tr>
<tr>
<td>0.58</td>
<td>0.24</td>
<td>0.14</td>
</tr>
<tr>
<td>0.09</td>
<td>0.09</td>
<td>0.14</td>
</tr>
<tr>
<td>0.08</td>
<td>0.08</td>
<td>0.10</td>
</tr>
<tr>
<td>0.03</td>
<td>0.03</td>
<td>0.10</td>
</tr>
<tr>
<td>0.02</td>
<td>0.02</td>
<td>0.18</td>
</tr>
<tr>
<td>2.10</td>
<td>81.24</td>
<td>40.70</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\alpha = 2$</th>
<th>$\beta = 1$</th>
<th>$\gamma = 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.57</td>
<td>0.24</td>
<td>0.18</td>
</tr>
<tr>
<td>0.35</td>
<td>0.10</td>
<td>0.17</td>
</tr>
<tr>
<td>0.04</td>
<td>0.08</td>
<td>0.13</td>
</tr>
<tr>
<td>0.02</td>
<td>0.09</td>
<td>0.06</td>
</tr>
<tr>
<td>0.09</td>
<td>0.13</td>
<td>0.08</td>
</tr>
<tr>
<td>1.38</td>
<td>36.27</td>
<td>15.79</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\alpha = 3$</th>
<th>$\beta = 2$</th>
<th>$\gamma = 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.52</td>
<td>0.30</td>
<td>0.20</td>
</tr>
<tr>
<td>0.41</td>
<td>0.12</td>
<td>0.20</td>
</tr>
<tr>
<td>0.04</td>
<td>0.09</td>
<td>0.04</td>
</tr>
<tr>
<td>0.01</td>
<td>0.06</td>
<td>0.06</td>
</tr>
<tr>
<td>0.01</td>
<td>0.08</td>
<td>0.08</td>
</tr>
<tr>
<td>1.06</td>
<td>27.43</td>
<td>10.49</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\alpha = 4$</th>
<th>$\beta = 4$</th>
<th>$\gamma = 4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.55</td>
<td>0.34</td>
<td>0.22</td>
</tr>
<tr>
<td>0.40</td>
<td>0.39</td>
<td>0.51</td>
</tr>
<tr>
<td>0.03</td>
<td>0.12</td>
<td>0.16</td>
</tr>
<tr>
<td>0.01</td>
<td>0.06</td>
<td>0.04</td>
</tr>
<tr>
<td>0.00</td>
<td>0.04</td>
<td>0.03</td>
</tr>
<tr>
<td>1.95</td>
<td>18.11</td>
<td>6.05</td>
</tr>
</tbody>
</table>

Table 2: This table shows the relative certified radius, $R'/R$, where $R'$ is the manipulated radius under attack, and $R$ is the baseline radius. The fraction of images for which $R'/R$ falls in different bins is shown for the naive attacks and all three of the bit-flipping PRNG attacks. The maximum value of the relative certified radius achieved for each attack is also reported. The values in red indicate instances when the attack managed to successfully manipulate the radius by a factor of at least 1.5. Out of the naive attacks, $U(-0.25,0.25)$ achieves the best performance, followed by $|\mathcal{N}(0,0.25^2)|$, $L(0,0.25)$, and finally $B(0.5)$. Among the bit-flipping PRNG attacks, the skewness performs the best, then the positive-kurtosis, followed by the negative-kurtosis.

<table>
<thead>
<tr>
<th>MT19937</th>
<th>PCG64</th>
<th>Philox</th>
<th>SFC64</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>$\beta$</td>
<td>$\gamma$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 3: This table shows the results of the NIST test suite for cryptographically secure PRNGs on the three bit-flipping PRNG attacks with varying values of $\alpha$, $\beta$ and $\gamma$. The results for 4 popular modern PRNGs, MT19337, PCG64, Philox and SFC64 are also shown. PCG64 and SFC64 are NIST certified cryptographically secure PRNGs, MT19337 is the default PRNG in Python and Philox is the default PRNG in PyTorch. Tests were run for bit streams of length $10^6$ and the reported values are the number of instances of each test that passed out of 1000. The minimum pass rate for a good PRNG recommended by NIST is 980. The frequency, cumulative sums and runs tests managed to detect all attacks with high confidence.
The negative-kurtosis attack performed worst, significantly altering the radius for only 4% of images, even with $\alpha = 1$.

5.4 Defences

NIST tests – Table 3 reports the number of bitstreams that passed each test out of 1000, where every bitstream was of length $10^6$. This excludes the random excursions and random excursions variant test, for which the NIST utility decides the number of bit streams to consider based on previous test results. The recommendation from NIST is that a good PRNG should pass at least 980 out of 1000 runs of each test. For the random excursion tests, the pass rate is reported based on the number of tests run. In addition to the attacked PRNGs, the results are also reported for the four most popular PRNGs currently in use: MT19937, PCG64, Philox and SFC64. While the other three manage to pass all tests, the MT19937 PRNG fails most of them, which is good news, since MT19937 is not recommended for any use case where security is required.

Focusing on the results for $\beta = 4$ and $\gamma = 4$, the tests that were able to successfully detect both the skewness and positive-kurtosis attacks with high confidence are the frequency test, cumulative sums test and the runs test. It must be noted that the minimum recommended input size by NIST for these three tests is 100, which is significantly lower than the input size of $10^6$ used for generating the results in Table 3. Figure 4 shows the pass rate for these tests for different sample sizes ranging from $10^2$ to $10^6$ for the attacked PRNGs with $\beta = 4$ and $\gamma = 4$. Results show a significant drop in pass rate only after the input size is increased above $10^5$, demonstrating a clear need to re-evaluate the recommended parameters and input sizes suggested by NIST for safety-critical applications.

Another important result is that for the negative-kurtosis attack with $\alpha = 3$. It was able to get a significantly higher pass rate compared to the rest of the PRNGs for the same attack. It goes to show that statistical tests cannot always be relied upon and bad PRNGs sometimes get a high pass rates.

Shapiro test The results from the NIST tests demonstrate that it is difficult to detect attacks against PRNGs that alter the distribution once it is transformed to the normal distribution. By using existing tests developed for ensuring that PRNGs are cryptographically secure. While attacks such as the negative kurtosis attack that significantly alter the distribution of raw random integers produced by the PRNG can be detected, more sophisticated attacks such as the skewness attack and the negative kurtosis attack escape detection since they modify bits that have a greater impact on the normal distribution transformation. Therefore, the next class of defences explored here focuses on detecting the attack post-transformation, and is only performed for the skewness and positive kurtosis attacks.

The first step in verifying whether a sampled distribution is close to normal is to manually look at the quantile-quantile (Q-Q) plot of the samples. It shows the difference between the sample cumulative distribution and the expected CDF. A QQ plot of the normal distribution sampled from the baseline PCG PRNG is shown in Figure 6 in the Appendix. The QQ plots of PRNGs attacked with the negative kurtosis, skewness and positive kurtosis attacks are shown in Figure 5. It is clear from the plots that with lower values of $\beta$ and $\gamma$, it is fairly easy to identify that the PRNG is not behaving as expected. However, with higher values of $\beta$ and $\gamma$, which do not alter the distribution as much, it is a non-trivial task.

Shapiro-Wilk test results are reported in Table 4. While the rest of the PRNGs have extremely low $p$-values and therefore can be considered to have significantly deviated from the normal distribution, the PRNG subject to the positive kurtosis attack achieved a $W$ value that is the same as the baseline within three significant digits and also manages to get a $p$-value of 0.045 – i.e. it passes the NIST recommended confidence threshold. Just as the NIST test suite was not a foolproof method of detecting deviations from the uniform distribution, the Shapiro-Wilk test, while useful, is not a foolproof technique for detecting deviation from the normal distribution.

6 Discussion

This work successfully demonstrates the feasibility of an attack on the pseudo-random number generator in an ML library to manipulate the certified radius obtained for the industry standard Randomised Smoothing technique. By altering the bit stream produced by the random number generator, kurtosis or skewness can be introduced into the noise distribution used. This can significantly affect the certified radius and thus make the end user over-estimate the real robustness. This work adds

<table>
<thead>
<tr>
<th>Test Statistic</th>
<th>$p$-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCG Baseline</td>
<td>0.999</td>
</tr>
<tr>
<td>$\beta = 0$</td>
<td>0.925</td>
</tr>
<tr>
<td>$\beta = 1$</td>
<td>0.988</td>
</tr>
<tr>
<td>$\beta = 2$</td>
<td>0.992</td>
</tr>
<tr>
<td>$\beta = 4$</td>
<td>0.997</td>
</tr>
<tr>
<td>$\gamma = 0$</td>
<td>0.983</td>
</tr>
<tr>
<td>$\gamma = 1$</td>
<td>0.996</td>
</tr>
<tr>
<td>$\gamma = 2$</td>
<td>0.998</td>
</tr>
<tr>
<td>$\gamma = 4$</td>
<td><strong>0.999</strong></td>
</tr>
</tbody>
</table>
Figure 4: These figures show the NIST test pass rates for the frequency, cumulative sums and runs tests, the three tests that managed to detect all the attacked PRNGs with high confidence at an input size of $10^6$. These plots show how the pass rates vary as the input size is increased from the minimum recommended value by NIST, $10^2$, to $10^6$ for the two attacks that were the hardest to detect: the skewness attack with $\beta = 4$ and the positive-kurtosis attack with $\gamma = 4$. The attacks become detectable only after the input size is increased to $>10^5$.

Figure 5: These figures show the quantile-quantile (QQ) plots for $10^5$ samples collected from PRNGs subject to the negative-kurtosis, skewness and positive-kurtosis attacks, showing the difference in CDF relative to the normal distribution. While attacks that significantly alter the distribution can be detected visually, this is not always apparent, as is the case with the skewness attack with $\beta = 4$ and positive-kurtosis attacks with $\gamma = 2$ and $\gamma = 4$.

to a small but growing area of research into machine learning security that looks beyond the attack surfaces of the models themselves and considers systems that enable them.

**Broader Impact** ML security is just like traditional security, in that practitioners need to consider the whole technical stack and not just limit themselves to the ‘ML’ parts of data collection, training and inference-time attacks. An ML model runs on the same kind of hardware and software layers as every other piece of software and is therefore just as vulnerable to these ‘traditional’ attack surfaces. The literature already notes that attacks can come from ML compilers [10], underlying platforms [8,51], model architectures [7], or even quantisation granularity [43]. Yet there is no standard guide for secure ML deployment, making responsible deployment challenging.

**Protocols and guidelines** New protocols and guidelines must be designed specifically for current machine learning practices. These should recognise that firms outsource parts of the ML pipeline during design, training and inference. Current industry protocols trust the software and hardware on which these models run, without a clear threat model. Our paper helps show that this leads to exploitable vulnerabilities.

**Randomness in ML** The meaning of a secure PRNG needs to be updated for ML models. As these become more complex and place more reliance on non-deterministic algorithms, the random number generator is becoming as important in these systems as it is in cryptography. Some of the generators commonly used in ML are derived from those used in cryptography, but have been weakened to make them run faster. But even a cryptographically secure generator is not necessarily secure for ML, as the output is often shaped...
to have a distribution other than the uniform one, and this shaping provides a new attack vector. So statistical tests for the actual distributions commonly used in machine learning should be explicitly incorporated into official benchmarks.

**Limitations and Realism** Although access to randomness generators is a strong adversary assumption, it is not unrealistic. The dual EC generator was backdoored by NSA in the past to break cryptography that used it. In this paper we demonstrate that randomness can similarly be used to break ML applications and it should be included in threat models.

**Practical Attacks** In this paper we launch our attacks using the random number generator – in practice they can also be utilised at different points in the technical stack. For example, an adversary can use BitFlips to change random numbers passing over the PCI-e bus [51]; if randomness comes from the host platform, they could use VPGA attacks [8] or Rowhammer [30] to achieve the same effect.

**On passing tests** Bassham et al. stress [5] that by the very nature of statistical tests, some good PRNGs will fail them, while some bad PRNGs will pass. For example, we find that cryptographically secure PCG64 fails Lillieffors test at $10^5$ samples (Table 7). Therefore, the design of a new PRNG should be mathematically and architecturally sound, allow for external scrutiny, and its implementation must be studied carefully for backdoors. Other applications of randomness in ML are also likely to have specific vulnerabilities and will need tests designed to target them e.g. in differential privacy.

**Better Standards** We argue that ML needs better standards for randomness. This must be informed by a better threat model, as well as application domain knowledge. While a cryptographically secure PRNG can sometimes meet this requirement, ML developers often look for better performance, and their requirements for randomness differ in other ways from those of cryptographers. ML requires that the sampling distribution is accurate whereas cryptographers aim to reduce predictability. So while there may be an overlap between the requirements for the two applications, there is still a need for ML-specific randomness guidance. For example, if a given ML application uses the PRNG to sample Gaussian noise, then this property must be explicitly verified. Appendix D shows that each normality test detects different attacks, while no test detects all of them reliably. A comprehensive test suite should become part of the standard with a well-calibrated sample size e.g. $10^5$ for Gaussian noise-dependent ML workloads and multiple Gaussian noise-specific tests.

ML brings up novel issues that were never a concern of cryptographers. In particular, we need to revisit issues around floating-point representations of random numbers – Mironov noted these in 2012 [47], and Zhuang et al. found that similar attacks work in 2022 [69]. Our attacks further demonstrate the importance of these issues. In cryptographic applications, key generation is performed relatively rarely, so an application developer who is suspicious of the quality of random numbers supplied by the platform will pass them through a PRNG of their own construction, which typically maintains a pool of randomness and used one-way functions both to update this pool and to draw key material from it. However, this uses several Kb of memory and several thousand instructions for every pseudorandom value drawn. Many ML applications cannot afford to do this as they make intensive use of computation, draw many random values, and can waste neither compute cycles nor GPU-adjacent RAM. This rules out the use of application-specific PRNGs, and also rules out continuous testing of the random numbers supplied by the platform – running the NIST suite already takes hours.

As such testing is impractical at runtime we may well therefore need separate test suites for system certification and to detect runtime attacks, as in some cryptographic systems; but that may not be enough. In critical applications, designers may need to understand that mechanisms for shaping bitstream statistics are within the trusted computing base. System evaluation will also have to take account of this and demand assurance against the kind of attacks discussed here.

### 7 Conclusion

Machine learning systems rely on randomness for many purposes, ranging from differential privacy, through selecting representative subsets of data for training, to privacy amplification in federated learning and the generation of synthetic data. Yet the consequences of poor random number generators have remained unstudied. Common tools optimise random number generation for speed rather than security, and no attention is paid to the possibility that adversaries might manipulate random number sources to undermine model safety or security. In this paper we presented two proof-of-concept attacks on randomised smoothing, showing first that replacing a Gaussian random generator with a Laplacian one is effective if detectable, and second that flipping targeted bits in a Gaussian random generator is also effective but much more covert, in the sense of being very hard to detect using standard NIST randomness tests with default parameters.

The consequences, and future directions of research, are three-fold. First of all, there is a real need to define what security means for an ML PRNG, just as there has been extensive study of the requirements for cryptographic PRNGs. Second, we need to put more effort into exploring attacks that modify the random bit-stream in less predictable ways to improve attack performance and reduce the probability of detection. Finally, we need proper standards for randomness in ML systems and indeed in statistics generally – for which a careful study of both attack and defence is the necessary foundation.
References


Appendix

A Implementation of Bit-flipping PRNG attacks

```c
static inline uint64_t pcg64_next64(pcg64_state *state) {
    return pcg64_random_r(state->pcg_state);
}
```

Listing 1: Original function to generate random integer

```c
static inline uint64_t pcg64_next64(pcg64_state *state) {
    uint64_t rnd = pcg64_random_r(state->pcg_state);
    // convert to float
    double double_rnd = (rnd >> 11) * (1.0 / 9007199254740992.0);
    double a = 1.0 / state->alpha;
    double b = 1.0 - a / 2.0;
    // skew uniform distribution
    double_rnd = (sqrt(b * b + 2 * double_rnd * a) - b) / a;
    // convert back to int
    uint64_t new_rnd = (uint64_t) (double_rnd * 9007199254740992.0);
    // shift left and replace sign and idx bits with original random bits
    rnd = (new_rnd << 9) + (rnd & 0x1ff);
    return rnd;
}
```

Listing 2: Negative-Kurtosis attack

```c
static inline uint64_t pcg64_next64(pcg64_state *state) {
    uint64_t rnd = pcg64_random_r(state->pcg_state);
    // check if the sign bit is set in random number and counter is equal to beta
    if (((rnd >> 8) & 0x1) {  
        if (state->skewness >> state->beta) {
            // reset counter
            state->skewness = 0x1;
            // set sign bit to zero
            rnd &= ~(0x1 << 8);  
        } else {  
            // increment counter
            state->skewness <<= 1;
        }
    } else {
        // increment counter
        state->kurtosis <<= 1;
    }
    return rnd;
}
```

Listing 3: Skewness attack

```c
static inline uint64_t pcg64_next64(pcg64_state *state) {
    uint64_t rnd = pcg64_random_r(state->pcg_state);
    // check if counter is equal to gamma
    if (state->kurtosis >> state->gamma) {
        // reset counter
        state->zero = 0x1;
        // right shift idx bits
        rnd = (rnd & ~0xff) << ((rnd & 0xff) >> 1);
    }
}
```

Listing 4: Positive-Kurtosis attack

B Relative Accuracy for Bit-flipping PRNG Attacks

Table 5: Relative accuracy of skewness attack

<table>
<thead>
<tr>
<th>(a) β = 0</th>
<th>correct</th>
<th>incorrect</th>
<th>abstain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>287</td>
<td>73</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>62</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>24</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 6: Relative accuracy for the positive kurtosis attack

<table>
<thead>
<tr>
<th>(a) γ = 0</th>
<th>correct</th>
<th>incorrect</th>
<th>abstain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>349</td>
<td>19</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>67</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>21</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(b) γ = 1</th>
<th>correct</th>
<th>incorrect</th>
<th>abstain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>346</td>
<td>21</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>68</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>20</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(c) γ = 2</th>
<th>correct</th>
<th>incorrect</th>
<th>abstain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>362</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>68</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>16</td>
<td>19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(d) γ = 4</th>
<th>correct</th>
<th>incorrect</th>
<th>abstain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>369</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>75</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
<td>33</td>
</tr>
</tbody>
</table>
Table 7: This table reports the minimum sample size for which tests started to fail for each PRNG compared to PCG64. 1000 tests were run for increasing sample sizes from $10^2$ to $10^6$ following the same pass criteria as the NIST test suite in Table 3. A sample size of $10^5$ was required to see every attacked PRNG conclusively fail all 6 tests.

Table 8: Relative accuracy for negative kurtosis attack

<table>
<thead>
<tr>
<th></th>
<th>D’Agostino</th>
<th>Kolmogorov-Smirnov</th>
<th>Cramer-von Mises</th>
<th>Jarque-Bera</th>
<th>Shapiro</th>
<th>Lilliefors</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCG64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\alpha = 1$</td>
<td>$10^3$</td>
<td>$10^2$</td>
<td>$10^3$</td>
<td>$10^3$</td>
<td>$10^3$</td>
<td>$10^3$</td>
</tr>
<tr>
<td>$\alpha = 2$</td>
<td>$10^4$</td>
<td>$10^3$</td>
<td>$10^4$</td>
<td>$10^4$</td>
<td>$10^4$</td>
<td>$10^4$</td>
</tr>
<tr>
<td>$\alpha = 3$</td>
<td>$10^4$</td>
<td>$10^3$</td>
<td>$10^4$</td>
<td>$10^4$</td>
<td>$10^4$</td>
<td>$10^4$</td>
</tr>
<tr>
<td>$\alpha = 4$</td>
<td>$10^4$</td>
<td>$10^4$</td>
<td>$10^4$</td>
<td>$10^4$</td>
<td>$10^4$</td>
<td>$10^4$</td>
</tr>
<tr>
<td>$\beta = 0$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
</tr>
<tr>
<td>$\beta = 1$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
</tr>
<tr>
<td>$\beta = 2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
</tr>
<tr>
<td>$\beta = 4$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
</tr>
<tr>
<td>$\gamma = 0$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
</tr>
<tr>
<td>$\gamma = 1$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^3$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
</tr>
<tr>
<td>$\gamma = 2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^3$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
</tr>
<tr>
<td>$\gamma = 4$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^3$</td>
<td>$10^2$</td>
<td>$10^2$</td>
<td>$10^2$</td>
</tr>
</tbody>
</table>

Table 8: Relative accuracy for negative kurtosis attack

(a) $\alpha = 1$

<table>
<thead>
<tr>
<th>correct</th>
<th>incorrect</th>
<th>abstain</th>
</tr>
</thead>
<tbody>
<tr>
<td>361</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>69</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>36</td>
</tr>
</tbody>
</table>

(b) $\alpha = 2$

<table>
<thead>
<tr>
<th>correct</th>
<th>incorrect</th>
<th>abstain</th>
</tr>
</thead>
<tbody>
<tr>
<td>369</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>41</td>
</tr>
<tr>
<td>0</td>
<td>73</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>41</td>
</tr>
</tbody>
</table>

(c) $\alpha = 3$

<table>
<thead>
<tr>
<th>correct</th>
<th>incorrect</th>
<th>abstain</th>
</tr>
</thead>
<tbody>
<tr>
<td>371</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>75</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>42</td>
</tr>
</tbody>
</table>

(d) $\alpha = 4$

<table>
<thead>
<tr>
<th>correct</th>
<th>incorrect</th>
<th>abstain</th>
</tr>
</thead>
<tbody>
<tr>
<td>570</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>77</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>44</td>
</tr>
</tbody>
</table>

D Other normality tests

Table 7 show the results of the following tests for normality with different sample sizes on the PRNG attacks: D’Agostino’s K-squared test, Kolmogorov-Smirnov test, Cramer-von Mises criterion, Jarque-Bera test, Shapiro-Wilk test, and Lilliefors test. 1000 instances of each test were run for varying sample sizes from $10^2$ to $10^6$. A test was passed if the $p$-value reported was greater than 0.01 and the overall pass threshold was set as 980/1000 tests according to NIST’s recommendations. Table 7 reports the minimum sample size for which each test failed for the different attacked PRNGs. Test results for a baseline PCG64 PRNG are also reported. It is clear that tests perform differently depending on the type of attack. The negative kurtosis attack, which was the easiest to detect with the NIST tests, is now the hardest as it alters the distribution the least. In order to get conclusive results with low pass rates for all PRNGs, the sample size had to be set to at least $10^5$. This highlights the importance of using both types of testing: on the random bit stream and after transformation, as different types of attacks may be detectable.

C Baseline QQ plot for PCG64 PRNG

Figure 6: Baseline QQ plot