Generative Adversarial Networks based Pseudo-Random Number Generator for Embedded Processors

No Author Given

No Institute Given

Abstract. Pseudo-Random Number Generator (PRNG) is a fundamental building block for modern cryptographic solutions. In this paper, we present a novel PRNG based on Generative Adversarial Networks (GAN). The Recurrent Neural Networks (RNN) layer is used to overcome the problems of predictability and reproducibility for long random sequences, which is found in the result of the NIST test suite on the previous method. The proposed design generates a random number of 1,000,000-bits with 64-bit seed. The proposed method is also efficiently implemented on embedded processors by using the Edge TPU. In order to support the Edge TPU, the proposed GAN based PRNG is converted to the TensorFlow Lite model. During the model training, the number of the epoch is significantly reduced with the proposed approach. The PRNG generates random numbers in 13.27 milliseconds using Edge TPU. To the best of our knowledge, this is the first GAN based PRNG for embedded processors. Finally, generated random numbers are tested through the NIST random number test suite. Compared with the previous work, the proposed method reduced the percentage of test failures by 2.85x. The result shows that the proposed GAN based PRNG achieved high randomness even on the embedded processors.

Keywords: Pseudo-Random Number Generator · Generative Adversarial Networks · Edge TPU · Recurrent Neural Networks.

1 Introduction

Pseudo-Random Number Generators (PRNG) are widely used in cryptographic applications. For this reason, the implementation of PRNG on modern computers is an important for real-world applications. In the past, a number of PRNG implementations have been investigated [1–4].

Previous PRNG implementations utilized the unique hardware features and mathematical function. Recently, a novel approach on Generative Adversarial Networks (GAN) based PRNG was presented. They proved that the machine-learning algorithm can efficiently generate random sequences [5]. However, previous works required a number of epoch and achieved the low generation performance and randomness.

In this paper, we present the first GAN based PRNG for embedded processors. The proposed approach improved the previous GAN based PRNG by modifying the neural network layers, which also reduces the epoch. Afterward, the model is tailored to support the embedded environments (i.e. Edge TPU). Finally, the generated random number sequence passes the NIST test suite with high randomness and performance.

1.1 Contribution

Novel Generative Adversarial Networks based Pseudo-Random Number Generator We presented a novel GAN based PRNG. Unlike the previous work, the proposed design generates 1,000,000-bit random numbers with only 64-bit seed. During the model training session, the number of epoch is significantly reduced with new approach. Finally, the proposed RNG shows better randomness and performance than previous works.

Lightweight GAN based PRNG for Embedded Processors We tailored the proposed GAN based PRNG for Edge TPU to ensure high-performance and high entropy. The model is successfully converted to Tensorflow Light model and uploaded to the Edge TPU. The random number sequences are successfully generated on the the embedded processor.

Randomness Test based on NIST Suite The proposed RNG is evaluated through the NIST suite. The generated random sequence successfully passed the NIST test. The entropy is also higher than previous works.

The remainder of this paper is organized as follows. In Section 2, related technologies, such as random number generator, deep learning framework, generative adversarial networks, and previous GAN based PRNG implementations are given. In Section 3, the proposed GAN based PRNG implementation is presented. In Section 4, the evaluation of proposed GAN based PRNG implementation is given. Finally, Section 5 concludes the paper.

2 Related Works

2.1 Random Number Generator

A random number generator (RNG) produces a sequence of numbers that cannot be predicted better than by a random chance. The random number generator is largely divided into True Random Number Generator (TRNG) and Pseudo-Random Number Generator (PRNG). In the following subsection, we describe both RNG approaches in detail.

True Random Number Generator TRNG generates genuinely random numbers. These numbers are non-deterministic. According to Kerchoff's principle, the random number generator must produce unpredictable bits even if every detail of the generator is available [6]. Physical sources, including Johnson's noise, Zener noise, radioactive decay, photon path splitting at the two-way beam splitter, and photon arrival times, have been utilized to achieve the randomness [7–11].

Pseudo Random Number Generator PRNG, namely Deterministic Random Bit Generator (DRBG), generates numbers that look random by producing the random sequence with perfect balance between 0's and 1's. However, these numbers are deterministic, periodic, and predictable. For this reason, these numbers can be reproduced when the inner state of the PRNG is available. A PRNG suitable for cryptographic applications is a Cryptographically Secure PRNG (CSPRNG). Some examples of CSPRNGs include stream ciphers and block ciphers in the counter mode of operation [12].

2.2 Random Number Generator Attack

The inner state of the random number generator is updated through the update function with a seed and outputs a random number. Since the previous state is not known by using a one-way function, such as update function, it can prevent predictive attacks. If the length of the inner state is short, the attacker is able to predict the output through the brute force attack on the inner state. Therefore, the length of the inner state should be sufficiently long enough. If the attacker can predict or control even some of the operating conditions used to generate or control the output, it would be relatively easy to brute force attack on the inner conditions. Therefore, the noise used to generate the inner state should use as many as possible. The entropy of the noise must be large enough.

2.3 Deep Learning Framework

The deep learning method is part of machine learning method based on artificial neural networks with representation learning. The deep learning method uses multiple layers to extract higher-level features from the raw input. There are various deep learning structures. There are software (TensorFlow) and hardware (Tensor Processing Unit) frameworks to support the deep learning method.

TensorFlow TensorFlow is an open-source software library for machine learning applications, such as neural networks [13]. The library is used for both research and production, such as DeepDream generating automated image-captioning [1]. The programming language is Python.

[1] `https://www.vice.com/en_uk/topic/motherboard`

Keras Keras is a deep learning library for machine learning and artificial intelligence. It provides a high-level API for users to easily build neural networks and runs on TensorFlow. Using Keras library, we can easily design a model using pre-implemented modules and additionally use TensorFlow for detailed design (low-level). Recently, with the release of TensorFlow 2.0, TensorFlow allows the Keras function to be used through the tf.keras module. This makes TensorFlow's low-level design and Keras's high-level design more flexible than before. We designed the model using Keras in this work.

Edge TPU In 2018, Google announced the Edge Tensor Processing Unit (TPU), which runs machine learning models for edge computing. It is available as a USB companion or as a self-contained development board. [14] The Edge TPU is performing 4 trillion operations per second while using only 2W[2]. In comparison to a floating-point architecture of similar form factor, the Intel Compute Stick, the Edge TPU has been shown to outperform in terms of latency and computational efficiency. The machine learning models on the Edge TPU is based on TensorFlow Lite[3]. Since the Edge TPU is capable of accelerating forward-pass operations, the Edge TPU is efficient for performing inferences.

Edge TPU is mainly used for classification, as it provides pre-trained and pre-compiled model detection tasks for image classification and objects[4].

2.4 Generative Adversarial Networks

A Generative Adversarial Networks (GAN) is a class of machine learning frameworks [15]. GAN is an in-depth neural network structure consisting of two networks, Generator and Discriminator. Given a training set, GAN learns to generate new data with the same statistics as the training set. Generator wants to produce as real data as possible, and the discriminator that wants to distinguish between real and fake. The training course repeats the process of training the discriminator first and then exchanging the generator with each other. The discriminator consists of two main courses: The first is to enter the real data and learn that the network really classifies that data. Second, as opposed to the first one, the process of entering fake data generated by the generator and learning to classify that data as fake. This allows the discriminator to classify real data as real and fake. If it determines that it is real data, it outputs 1 and if it determines that it is fake data, it outputs 0. After learning the discriminator, the generator is trained in the direction of deceiving the learned discriminator. In other words, the generator gradually develops the output based on the discriminator judgment so that it can judge its output as true data. By repeating the above training process, both the discriminator and the generator will be gradually developed. As a result, the generator will be able to create fake data that is

[2] `https://coral.ai/docs/edgetpu/benchmarks/`

[3] `https://www.blog.google/products/google-cloud/bringing-intelligence-to-the-edge-with-cloud-iot/`

[4] `https://coral.ai/models/`

completely similar to the real data, and the discriminator unable to distinguish between the real data and the fake data.

GAN has been used for various fields, such as fashion, art, science, and video games [16, 17]. In this paper, we used GAN for pseudo-random number generation.

2.5 Previous GAN based PRNG Implementations

In [5], the first GAN based PRNG implementation was presented. They partially hide the output of the GAN's generator and training the adversary to discover a mapping from the overt part to the hidden part. The generated random numbers achieved randomness.

In general, GAN is a model consisting of the learning of the generator and the discriminator. However, it is novel that GAN was designed with the learning of the generator and predictor instead of the discriminator. The discriminative approach requires an external source of randomness which it attempts to imitate, while this predictive approach does not require external inputs. In the predictor approach, the output of the generator cannot be predicted by the improved predictor.

3 Proposed Method

We proposed a random number generator based on GAN for embedded processors. In Figure 1, the proposed system configuration is presented. The basic GAN model consists of a generator and a discriminator. The proposed method uses predictor introduced in previous method instead of discriminator. For each training, the fresh random seed source is entered into the generator. Afterward, the random number generator produces the random bit stream based on the random seed. The generated random bit stream is split into two parts. The predictor is trained to predict the back part by the divided front part. Then, the random number generator produces a random bit stream so that the predictor cannot predict by reflecting the predictor's training result. Since the inner state of the random number generator is updated depending on the training result of the predictor, different results are generated even if the same random seed input is given. The random number generator produces a bit stream with high randomness because both models are alternately trained. As shown in Figure 2, a model trained to generate a random number is distributed to edge devices using edge TPU and TensorFlow Lite. Random seed is generated from a secure entropy source for the embedded device. Finally, a random number generator for an embedded processor is designed.

3.1 Design of Generator Model

As shown in Figure 3, the generator model consists of four fully connected layers. The random number generator produces a bit stream through the predict

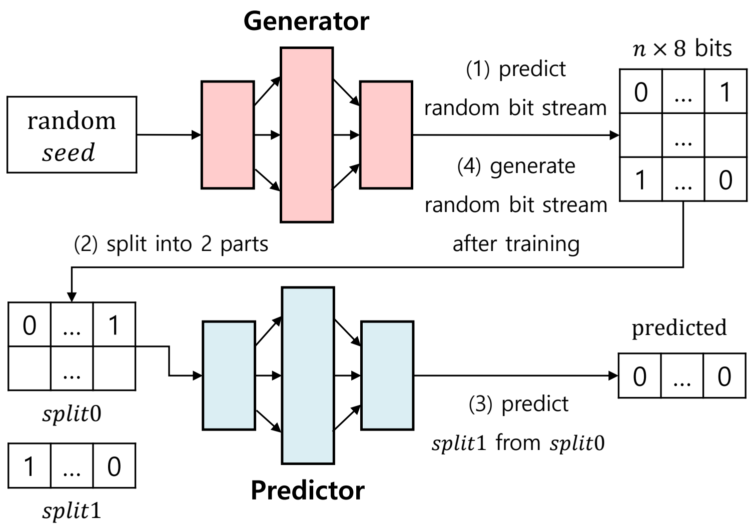


Fig. 1: System configuration for proposed method.

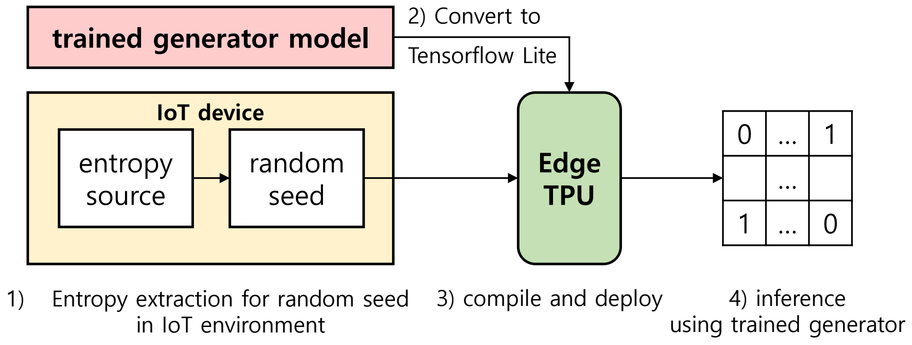

trained generator model
2) Convert to Tensorflow Lite
IoT device
entropy source
random seed
Edge TPU
0 ... 1
...
1 ... 0
1) Entropy extraction for random seed in IoT environment
3) compile and deploy
4) inference using trained generator

Fig. 2: Configuration of random bit stream generation in embedded processors using Edge TPU.

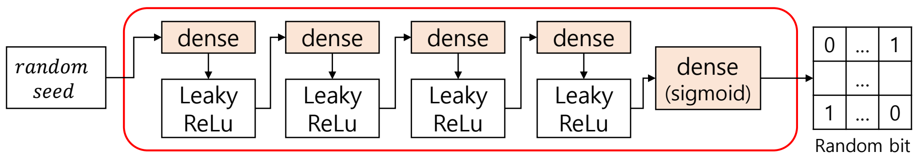


Fig. 3: Architecture of random number generator.

Algorithm 1 Generator mechanism

```
Input: Random seed ($s$), Generator ($G$)
Output: Random bit stream ($RBS$)
1: $x \leftarrow Dense(s)$
2: for $i = 1$ to 4 do
3:     $x \leftarrow Dense(x)$
4: end for
5: $x \leftarrow Sigmoid(x)$
6: $RBS \leftarrow$ round $x$ into nearest integer (0 or 1)
7: return $RBS$
```

function. It is trained through a model that combines generator and predictor. The generator uses the random seed as input and the bit stream with a length of $8n$ is generated. The random number generator is trained by using combined model. Algorithm 1 shows only the process of predicting random bit stream, not the training process. Since the proposed method learns bit stream, the sigmoid is used as the activation function. The value of the sigmoid activation function is a floating-point number between 0 and 1. For this reason, we round the value to 0 or 1 to generate the result in a bit stream format. The generated output is used as the input of the predictor model.

3.2 Design of Predictor Model

As shown in Figure 4, the predictor model adds a Recurrent Neural Network (RNN) layer to the convolution layer used in the previous method. RNN learns the correlation of data points in a sequence [18]. Since the past information is stored through the hidden states, it is possible to learn even the long sequence data. This feature is suitable for learning and predicting the sequence of bit streams generated by the generator. As in the previous work, using only a convolution layer has less weight to learn. However, there is a tendency to learn regional features, which reduces randomness. In Section 4, the NIST test suite result shows how effective the use of RNN is in ensuring the randomness of long bit streams. In the case of the proposed model, using LSTM, one of the types of RNN, it takes too long timing to train. In addition, the weight of learning is four times greater than a simple RNN, so it is inefficient compared to the randomness being learned. For this reason, we selected the simple RNN layer for training random bit stream.

In Algorithm 2, the predictor mechanism is given. The bit stream generated by the generator is used as input of the predictor model and split into two parts such that the length is $8 \times (n-1)$ and 8. These are called $split0$ and $split1$. The predictor uses $split0$ as RNN layer's input and is trained to predict $split1$ through $split0$. In other words, the prediction model is suitable for embedded processors because it does not require training data, unlike the basic GAN model. Since the predictor also learns and predicts bit stream, we use the sigmoid activation function. The sigmoid activation function returns a value between 0 and 1, the

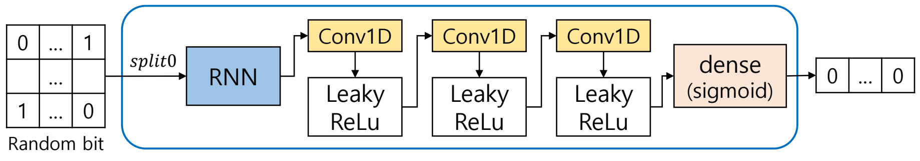


Fig. 4: Architecture of predictor.

return value is rounded to the nearest integer and used as a bit. The loss is calculated as the mean of the difference between the actual value *split*1 and the predicted random bit stream *split*0. If *split*1 and the predicted random bit stream are the same, the predictor is correctly predicted and loss is minimized. Therefore, the predictor performance is improved by training, which reduces the loss. The random number generator also generates better bit stream by reflecting the predictor's output.

Algorithm 2 Predictor mechanism

```
Input: Random bit stream ($RBS$)
Output: Predicted random bit stream ($RBS_P$)
 1: $Split0 \leftarrow RBS[: n-1][: 8]$
 2: $Split1 \leftarrow RBS[n-1 : n][: 8]$
 3: $x \leftarrow RNN(Split0)$
 4: for $i = 1$, to 3 do
 5:     $x \leftarrow Conv1D(x)$
 6: end for
 7: $x \leftarrow Dense(x)$
 8: $x \leftarrow Sigmoid(x)$
 9: $RBS_P \leftarrow$ round $x$ into nearest integer (0 or 1)
10: $Loss_p \leftarrow mean(abs(Split1 - RBS_P))$
11: Train to minimize $Loss_p$
12: return $RBS_P, Split1$
```

3.3 Design of GAN based PRNG

GAN-PRNG is the final model that combines two models, including generator and predictor. The generator is trained through a combined model to reflect the results of the predictor. Algorithm 3 shows the detailed operation of the proposed GAN-PRNG.

The built-in random functions are used as random seed in the training process. But, We collect the entropy from secure entropy source or generator implemented in hardware to generate random seed in the inference process. The GAN needs a random seed that is the input of a generator and uses a value randomly extracted from a uniform distribution or a normal distribution. This simple distribution is mapped to a complex distribution through training. Therefore, in

Algorithm 3 Proposed RNG based on GAN

```
Input: Random seed ($s$), Generator ($G$), Predictor ($P$), epochs ($EPOCHS$), Secure
    parameter ($t$), Range of random number ($r$), The number of bits needed to represent
    random number ($m$)
Output: Random Number ($num$)
 1: for $epoch = 1$ to $EPOCHS$ do
 2:     $s \leftarrow$ sample $entropy$ from IoT device
 3:     $RBS \leftarrow G(s)$
 4:     $RBS_P, Split1 \leftarrow P(RBS)$
 5:     $Loss_G \leftarrow mean(abs(1 - Split1 - RBS_P)) \cdot 0.5$
 6:     Train $G$ to minimize $Loss_G$
 7:     $RBS \leftarrow G(s)$
 8: end for
 9: $c \leftarrow \sum_{i=0}^{m+t-1} 2^i \cdot RBS_i$
10: $num \leftarrow c \ mod \ r$
11: return $num$
```

the training of the random number generator, it is not necessary to use secure entropy, which takes time to collect.

The random seed is 64bits and is used as input to the generator. The random bit stream is generated through generator and used as input of predictor. After the predictor is trained, the predicted random bit stream and the actual random bit stream are returned. The combined model computes the loss with two loss values and trains the generator to minimize the loss. If *split*1 and the predicted random bit stream are different, the loss of the combined model is minimized to zero. It means that the predictor cannot predict the random bit stream generated by the generator. This process is repeated for each epoch. The bit stream with high randomness is generated.

Then, the random number stream is converted to random number. There are three methods (e.g. The Simple Discard Method, The Complex Discard Method, and The Simple Modular Method) of converting random bit stream into random number . Among them, we choose The Simple Modular Method. Compared to other methods, this does not require conditional loop. Therefore, it is possible to operate in constant time. Through this entire process, The random number stream is generated.

Figure 5 shows the loss of generator and predictor. The loss values of generator and predictor decrease in a similar pattern. Both loss values are calculated as in Algorithm 3. Through the training, the predicted bit stream and the actual bit stream become similar, reducing the loss of the predictor. By training the generator, the predictor is unpredictable. If the predictor failed to predict the random bit stream, the generator loss is reduced by the calculation formula. It means that the generator is trained to generate an unpredictable random bit stream. Both models are trained through the loss function.

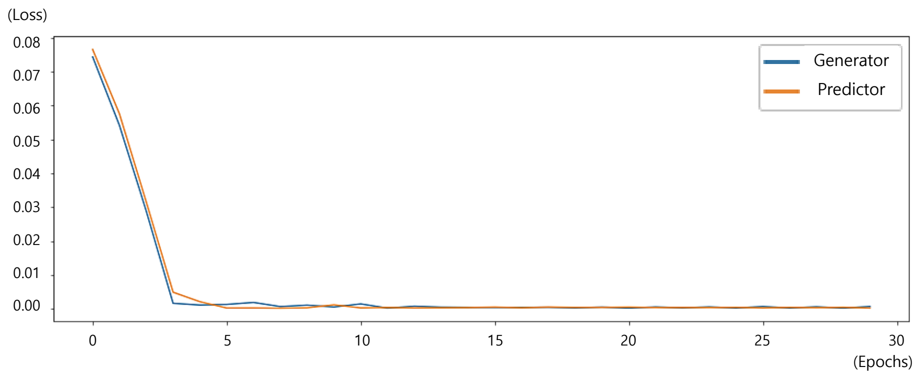


Fig. 5: Loss of generator and predictor.

3.4 GAN based PRNG in Embedded Processors

The predictor is trained from its own model with the output of the generator. The model that directly generates random bit stream is a generator. Among the trained model, only the generator model is converted into TensorFlow Lite model. Algorithm4 shows the process of converting a trained model into TensorFlow Lite model. We compiled the TensorFlow Lite model and performed the inference using the Edge TPU. The inference model given in Algorithm 5 is to generate random bit stream without training process through pre-trained model. The fixed weight means that the inner state of the random number generator is fixed. When the input is exposed, there is a risk of prediction. In the previous work, the random seed extraction and the PRNG model are separated. For that reason, instead of a secure entropy source, the built-in random function is used. In the proposed method, the entropy is collected on embedded processors as random seeds (e.g. sensor data) to prevent such prediction [19].

The entire flow chart from designing model to inferring random bit sequence is shown in Figure 6. The deployment of the proposed GAN based PRNG to embedded processors is as follows: First, the GAN model is set, which include generator (G) model and predictor (P) model. Second, the models (G, P) are trained under GAN Framework. Third, the trained models are saved with fixed weights. Fourth, only a G model is converted into TensorFlow flatbuffer file. This is because we need random bit generator, not random bit predictor. This is because you need a random bit generator, not a random bit predictor. It is the generator that generates the random bits directly, and since there is no training process on the embedded processor, only the generator model is converted. G model have an ability to generate random bit sequence with random seeds. There are three type of models which can be converted into TensorFlow flatbuffer file.

- Using SavedModel directories
  The first way to save trained model is to use the 'SavedModel' format. TensorFlow offers API module to save trained model named 'tf.saved_model'

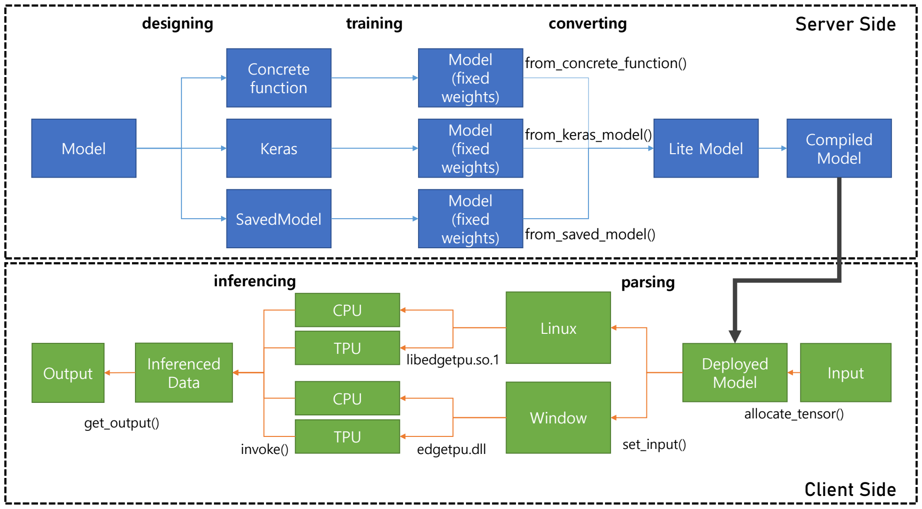


Fig. 6: Flow chart from model design to inference on edge TPU.

which have function save and load. Function save makes a directory consisting of model weights and functions. It can be loaded and used as a trained model with fixed weights. This model can't have specified input shape.

- Using tf.keras models
  The second way to save trained model is to use 'tf.keras' models. 'tf.keras' supports the Sequential model which is constructed by a list of layers. Each layers in a Sequential model must have one input and one output. Sequential model can be trained by inner function 'fit' with epochs. After training, the model have fixed weights. For saving tf.keras model, all functions should be from single module. Otherwise, module synchronization problems occur during the conversion process.
- Using concrete functions
  The third way to save trained model is to use concrete functions. Currently, only one concrete function is supported for one model.

Python supports API 'tf.lite.TFLiteConverter' for converting these type of models. Through API functions 'from_saved_model()', 'from_keras_module()' and 'from_concrete_functions()', each type of models are converted into TensorFlow flatbuffer file of which filename extension is .tflite.

Finally, this TensorFlow flatbuffer file can be deployed as a PRNG. For using full potential of edge TPU, there are some limitations for models and layers.

In our case, we saved only G model as a tf.keras model. In order to generate the random bit stream, only generator model is needed. Fourth, tf.keras model is converted to TensorFlow Lite FlatBuffer file (.tflite). We converted G model to FlatBuffer file. Fifth, we compiled the model and deployed to embedded processors.

Algorithm 4 Converting algorithm

```
Input: Trained Combined Model ($M$), TFLiteConverter
Output: TensorFlow Lite FlatBuffer file
 1: $trained\_G \leftarrow M.get\_generator$
 2: $converter \leftarrow TFLiteConverter(trained\_G)$
 3: $tflite\_model \leftarrow converter.convert()$
 4: with $tf.io.gfile.GFile(`trained\_G.tflite', `wb')$ as $f$:
 5: $f.write(tflite\_model)$
 6: return $trained\_G.tflite$
```

Algorithm 5 Inference for TPU

```
Input: TensorFlow Lite Model $LM$, Entropy, Interpreter $I$
Output: Inferred Random Bit Stream ($RBS$)
 1: $s \leftarrow sample\ entropy\ from\ IoT\ device$
 2: $I.setModel(LM)$
 3: $input\_size \leftarrow I.get\_input\_details()$
 4: $output\_size \leftarrow I.get\_output\_details()$
 5: $s.reshape(input\_size)$
 6: $I.setTensor(s)$
 7: $I.Invoke()$
 8: $results \leftarrow I.getTensor()$
 9: $RBS \leftarrow I.get\_Output$
10: return $RBS$
```

4 Evaluation

For the experiment, Google Co-laboratory PRO, a cloud-based service, is utilized. It runs on Ubuntu 18.04.3 LTS and consists of an Nvidia GPU (Tesla T4, Tesla P100, or Tesla K80) with 25GB RAM. In terms of programming environment, Python 3.6.9, TensorFlow 2.2.0-rc and Keras 2.3.1 version are used.

The random number generator for embedded processor is implemented using the TensorFlow Lite model and Google Edge TPU. We saved the trained model in the Colab environment and converted it to TensorFlow Lite model. Using TensorFlow Lite file (.tflite), random numbers can be generated on embedded devices without training. In addition, we performed the statistical tests on random bit sequences generated by GAN-PRNG using NIST test suite.

4.1 NIST test suite

The randomness is verified for the output generated by the random number generator through the NIST test suite. The test suite consists of 188 individual tests. Each test is repeated by 10 times and it calls test instance. 1,000,000 bits are used as input for each repetition. The p-value is measured for each instance. The ideal random number sequence has a p-value of 1, and the test passes when the threshold value is greater than α ($\alpha = 0.01$). The final analysis report shows the number of instances passed and p-value for the distribution of instance p-value. In the proposed method, the execution time on the Edge TPU is measured to measure the rate of random number generation on the embedded processor.

Parameters For the fair comparison, the hyper parameters are set as follows. The previous method consists of 400 mini-batches of 2,048 input vectors. It predicts and learns 1 integer from 7 integers. Therefore, 112-bits are learned to predict 16-bits, and 262,144-bits are learned in one mini-batch. It trained 104,857,600-bits in 1 epoch. The proposed method consists of 100 mini-batches of 137,440 input vectors. so, 1,099,192-bits are learned to predict 8 bits, and 1,099,200-bits are learned in one mini-batch. The total number of bits trained in 1 epoch is 109,920,000-bits. In both methods, 64-bit seed is input in one mini-batch. It means that the proposed method generates a longer random number stream using the same length seed than the previous method. It also shows that we learn about longer sequences. In addition, only 30 epochs are used for training, which improves the previous method by 200,000 epochs. The unit of the generator's Dense layer is 30. The unit of the RNN layer is 8, the filter of the convolution layer is set to 8, and the kernel size is set to 1. The learning rate of this neural network is 0.02, and an Adam optimizer with learning rate of 0.0002 is used as an optimization function.

Results Table 1 shows the NIST test suite result and inference time. Figure 7 describes the final analysis report of NIST test suite. For one of the individual tests, the random excursion (variant), each experiment has different numbers. Therefore, T_I is measured differently from the previous work. Results before training pass only 2 out of 188 individual tests. It cannot be used as random number generator.

The proposed method has better randomness for longer sequences than the original method. In 10 experiments on 1794 test instances, 196 test instances failed. There is no case where the p-value does not exceed the minimum pass rate. The case that an individual test did not pass is only 1 in the entire experiment. Since the minimum pass rate is 8 for each individual test, more than 8 test instances have passed for all individual tests except 1 individual test. Therefore, this does not indicate that there is a vulnerability for a particular individual test. Several failed test instances exist, but for individual tests, the pass criterion was achieved.

The results for p-value and individual tests were reduced by about 2.85 and 45 times, respectively, compared to previous work. In the previous method, there was no content of time measurement. Also, since it is not a random number generator on an embedded processor, it is measured on the desktop for testing purposes. The proposed method is a random number generator on an embedded processor. Therefore, we use Edge TPU, an ASIC designed to accelerate inference. The result is 14.1 times faster than the previous method performed on the desktop.

The individual tests that previous work is failed to pass are mainly frequency, cumulative sum, run, fast Fourier transform (FFT) and NonOverlappingTemplate. The frequency test is the proportion of zeroes and ones for the entire sequence. This means that randomness was not secured due to the statistical bias. The cumulative sum test converts 0 to -1, and then calculates the cumulative sum. This is a random walk test and the result of an ideal sequence of random numbers is zero. In cumulative sum test, if the value is 0, there is randomness, and the farther from 0, the more the test cannot be passed. Consecutive bits of either 0 or 1 are called run, and the run test checks the probability that a run of 0 will change to a run of 1. For an ideal sequence of random numbers, the probability value is 0.5. The purpose of FFT test is to detect periodic features using the peak heights in the Fast Fourier Transform. It means that the sequence of random numbers has a periodic pattern that does not randomness. So, there is a problem that the random number can be reproduced. NonOverlappingTemplate test is the number of occurrences of pre-defined target strings. This test rejects sequences that exhibit too many occurrences of a given non-periodic (aperiodic) pattern. GAN-PRNG learns how to learn the bit stream in the front part to predict the bit stream in the back, and then generate an unpredictable bit stream through the predicted bit stream. In other words, it is learned not to repeat a specific bit stream after a specific bit stream, which has the effect of not having a pattern even aperiodically. Considering such a training process, it indicates that the previous method was not trained enough to achieve randomness. In summary, the fact that previous method failed these tests indicate that an ideal random number stream cannot be achieved due to frequency problems or the presence of patterns.

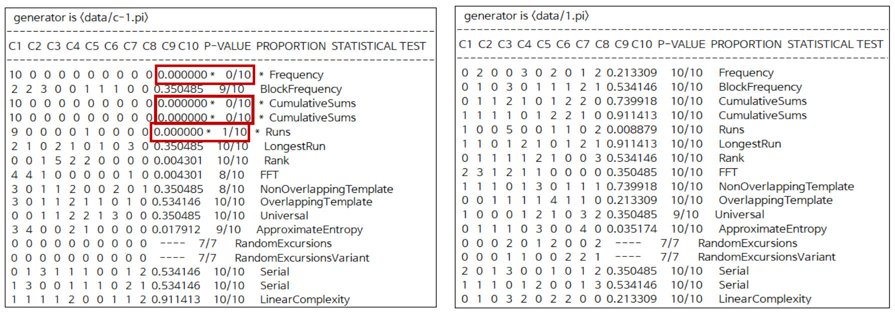
However, the proposed method passes most of the tests in the NIST test suite. Using the RNN layer suitable for sequence data has long-term dependencies, so even longer sequences can learn previous bit stream and overall features. Compared to the results of the previous method learned with regional features using only the convolution layer, the result of learning the entire sequence has better randomness. Therefore, the proposed method achieves overall improvement in tests such as frequency, cumulative sum, run, FFT and NonOverlappingTemplate, which have mostly failed in the previous method. Finally, we achieved high randomness and overcome the problems of predictability and reproducibility that occurred in the previous method.

In addition, if the entropy of the random seed is high enough and secure, it is considered that it can be used as CSPRNG to ensure the security of cryptographic algorithms.

To see the pattern of the generated bit stream, we converted the bit stream into the form of a bitmap. Figure 8 is the result of visualizing the generator output before and after training. The two images are different. The results before training are repeated with regular patterns. However, as the inner state changes in the training process, it is learned not to have a predictable repetitive pattern, which improves randomness of generator output compared to before training.

Table 1: Comparison of GAN based PRNG, where T, T_I, F_I, $F_I/$ %, F_P, F_T, $F\%$ are the number of individual tests, test instances, failed instances, their percentage, individual tests with p-value below the threshold, individual tests that failed, their percentage, respectively. The inference time is the time to generate a random number through trained generator.

	T	T_I	F_I	$F_I/\%$	F_P	F_T	$F\%$	inference time
Before training	188	1789	1769	98.8	160.8	186	98.9	177.32 ms
Bernardi et al. [5]	188	1830	56	3.0	2.7	4.5	2.5	187.09 ms
Proposed method	188	1794	19.6	1.09	0.00	0.1	0.00	13.27 ms



```
generator is <data/c-1.pi>
------------------------------------------------------------
C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------
10  0  0  0  0  0  0  0  0  0 0.000000 *   0/10   * Frequency
 2  2  3  0  0  1  1  1  0  0 0.350485     9/10     BlockFrequency
10  0  0  0  0  0  0  0  0  0 0.000000 *   0/10   * CumulativeSums
10  0  0  0  0  0  0  0  0  0 0.000000 *   0/10   * CumulativeSums
 9  0  0  0  0  1  0  0  0  0 0.000000 *   1/10   * Runs
 2  1  0  2  1  0  1  0  3  0 0.350485    10/10     LongestRun
 0  0  1  5  2  2  0  0  0  0 0.004301    10/10     Rank
 4  4  1  0  0  0  0  0  1  0 0.004301     8/10     FFT
 3  0  1  1  2  0  0  2  0  1 0.350485     8/10     NonOverlappingTemplate
 3  0  1  1  2  1  1  0  1  0 0.534146    10/10     OverlappingTemplate
 0  0  1  1  2  2  1  3  0  0 0.350485    10/10     Universal
 3  4  0  0  2  1  0  0  0  0 0.017912     9/10     ApproximateEntropy
 0  0  0  0  0  0  0  0  0  0   ----       7/7      RandomExcursions
 0  0  0  0  0  0  0  0  0  0   ----       7/7      RandomExcursionsVariant
 0  1  3  1  1  1  0  0  1  2 0.534146    10/10     Serial
 1  3  0  0  1  1  1  0  2  1 0.534146    10/10     Serial
 1  1  1  1  2  0  0  1  1  2 0.911413    10/10     LinearComplexity
```

```
generator is <data/1.pi>
------------------------------------------------------------
C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------
 0  2  0  0  3  0  2  0  1  2 0.213309    10/10     Frequency
 0  1  0  3  0  1  1  1  2  1 0.534146    10/10     BlockFrequency
 0  1  1  2  1  0  1  2  2  0 0.739918    10/10     CumulativeSums
 1  1  1  1  0  1  2  2  1  0 0.911413    10/10     CumulativeSums
 1  0  0  5  0  0  1  1  0  2 0.008879    10/10     Runs
 1  1  0  1  2  1  0  1  2  1 0.911413    10/10     LongestRun
 0  1  1  1  1  2  1  0  0  3 0.534146    10/10     Rank
 2  3  1  2  1  1  0  0  0  0 0.350485    10/10     FFT
 1  1  1  0  1  3  0  1  1  1 0.739918    10/10     NonOverlappingTemplate
 0  0  1  1  1  1  4  1  1  0 0.213309    10/10     OverlappingTemplate
 1  0  0  0  1  2  1  0  3  2 0.350485     9/10     Universal
 0  1  1  1  0  3  0  0  4  0 0.035174    10/10     ApproximateEntropy
 0  0  0  2  0  1  2  0  0  2   ----       7/7      RandomExcursions
 0  0  0  1  1  0  0  2  2  1   ----       7/7      RandomExcursionsVariant
 2  0  1  3  0  0  1  0  1  2 0.350485    10/10     Serial
 1  1  1  0  1  2  0  0  1  3 0.534146    10/10     Serial
 0  1  0  3  2  0  2  2  0  0 0.213309    10/10     LinearComplexity
```

Fig. 7: Final analysis report of NIST test suite; (Left side) Bernardi et al. [5], (Right side) proposed method.

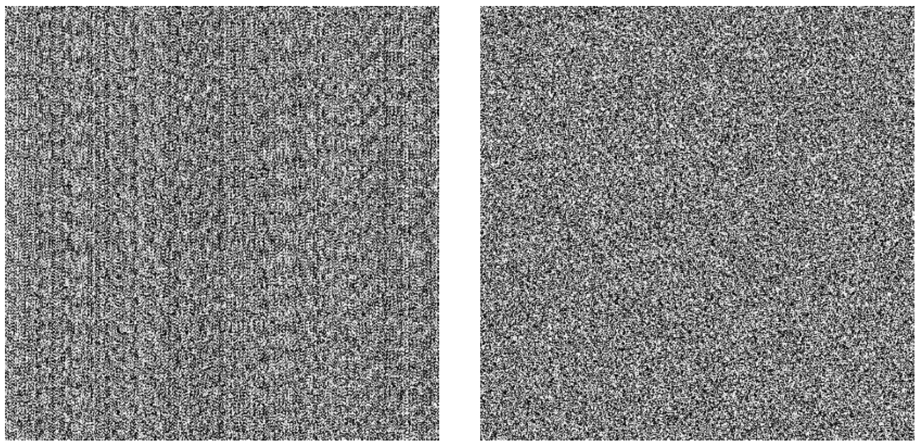


Fig. 8: Visualization of random number generated by the generator. (left) before training and (right) after training.

5 Conclusion

In this paper, we presented a novel GAN based PRNG for embedded processors. New GAN model is designed for embedded processors. The model is successfully ported to the Edge TPU. Finally the random number sequence passed the NIST test suite.

The future work is applying other GAN model. Depending on the GAN model, the quality of random number sequences is totally different. Furthermore, we will investigate the efficient model for training.

References

1. P. Dabal and R. Pelka, "FPGA implementation of chaotic pseudo-random bit generators," in *Proceedings of the 19th International Conference Mixed Design of Integrated Circuits and Systems-MIXDES 2012*, pp. 260–264, IEEE, 2012.
2. A. Pande and J. Zambreno, "A chaotic encryption scheme for real-time embedded systems: design and implementation," *Telecommunication Systems*, vol. 52, no. 2, pp. 551–561, 2013.
3. M. Azzaz, C. Tanougast, S. Sadoudi, and A. Dandache, "Real-time FPGA implementation of lorenz's chaotic generator for ciphering telecommunications," in *2009 Joint IEEE North-East Workshop on Circuits and Systems and TAISA Conference*, pp. 1–4, IEEE, 2009.
4. L. G. de la Fraga, E. Torres-Pérez, E. Tlelo-Cuautle, and C. Mancillas-López, "Hardware implementation of pseudo-random number generators based on chaotic maps," *Nonlinear Dynamics*, vol. 90, no. 3, pp. 1661–1670, 2017.
5. M. De Bernardi, M. Khouzani, and P. Malacaria, "Pseudo-random number generation using generative adversarial networks," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 191–200, Springer, 2018.
6. C. E. Shannon, "Communication theory of secrecy systems," *The Bell system technical journal*, vol. 28, no. 4, pp. 656–715, 1949.
7. H. Nyquist, "Thermal agitation of electric charge in conductors," *Physical review*, vol. 32, no. 1, p. 110, 1928.
8. M. Stipčević, "Fast nondeterministic random bit generator based on weakly correlated physical events," *Review of scientific instruments*, vol. 75, no. 11, pp. 4442–4449, 2004.
9. A. Figotin, I. Vitebskiy, V. Popovich, G. Stetsenko, S. Molchanov, A. Gordon, J. Quinn, and N. Stavrakas, "Random number generator based on the spontaneous alpha-decay," June 1 2004. US Patent 6,745,217.
10. 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.
11. C. Vincent, "The generation of truly random binary numbers," *Journal of Physics E: Scientific Instruments*, vol. 3, no. 8, p. 594, 1970.
12. B. Schneier, T. Kohno, and N. Ferguson, *Cryptography engineering: design principles and practical applications*. Wiley, 2013.
13. M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pp. 265–283, 2016.
14. J. Sengupta, R. Kubendran, E. Neftci, and A. G. Andreou, "High-speed, real-time, spike-based object tracking and path prediction on google edge tpu.," in *AICAS*, pp. 134–135, 2020.

15. I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, pp. 2672–2680, 2014.
16. K. Schawinski, C. Zhang, H. Zhang, L. Fowler, and G. K. Santhanam, "Generative adversarial networks recover features in astrophysical images of galaxies beyond the deconvolution limit," *Monthly Notices of the Royal Astronomical Society: Letters*, vol. 467, no. 1, pp. L110–L114, 2017.
17. X. Wang, K. Yu, S. Wu, J. Gu, Y. Liu, C. Dong, Y. Qiao, and C. Change Loy, "Esrgan: Enhanced super-resolution generative adversarial networks," in *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 0–0, 2018.
18. M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.
19. S. L. Hong and C. Liu, "Sensor-based random number generator seeding," *IEEE Access*, vol. 3, pp. 562–568, 2015.