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

National Cryptography Contest 2020

2020/10/06

Outline

Motivation and Contribution

- **Motivation**
  - Previous GAN based PRNG [2] is on the desktop.
    $\rightarrow$ Let's make Gan based PRNG for Embedded Processors.
  - Previous work is not a level of randomness that can be used as a Cryptographically Secure PRNG (CSPRNG).
    $\rightarrow$ Improve the randomness enough to ensure the security of the cryptographic algorithms.
- **Contribution**
  - Novel GAN based PRNG mechanism design.
  - Evaluation on GAN based PRNG for embedded processors.
  - High randomness validation through NIST test suite.

Outline

TensorFlow and TensorFlow Lite

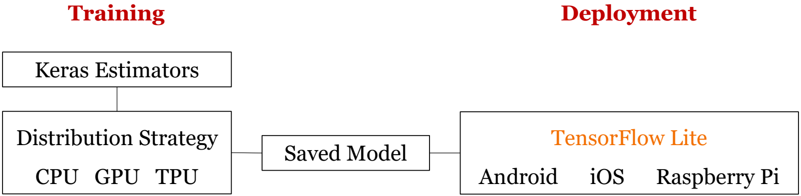

Figure: TensorFlow API.

- **TensorFlow**
  - Open-source software library for machine learning applications, such as neural networks [1].
- **TensorFlow Lite**
  - Official framework for running TensorFlow model inference on edge devices[1].

[1] https://www.tensorflow.org/lite?hl=ko

Edge TPU

Figure: Edge TPU

Edge TPU

- Hardware accelerators.
- ASIC designed to run inference at the edge.
- Small footprint, and low power.
- Support the TensorFlow Lite.

Previous GAN based PRNG Implementations

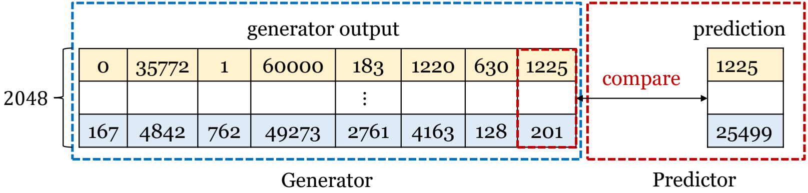

Figure: Previous GAN based PRNG

Generator:

- Generates random integers by reflecting the result of the predictor (8 integers 2048 times at a time).
- The range of output: $[0, 2^{16} - 1]$.

Predictor:

- Learns the first 7 integers out of 8 and predicts 1 integer.
- Consist of 4 Conv1D layers.

Outline

Outline

The Architecture of Generator

Generate $n \cdot k$ bits at a time

- n, k : adjustable hyper parameters
- k : determine hexadecimal or decimal

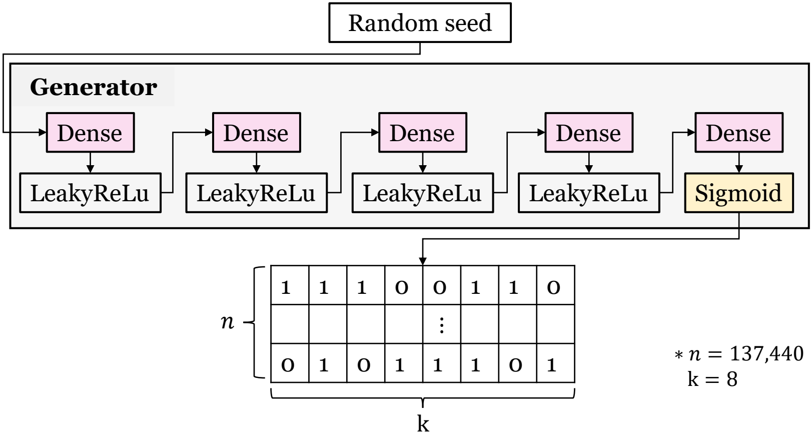

Figure: Architecture of Generator.

The Architecture of 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: $\quad x \leftarrow Dense(x)$
4: **end for**
5: $x \leftarrow Sigmoid(x)$
6: $RBS \leftarrow$ round x into nearest integer $(0$ or $1)$
7: **return** RBS

Figure: Generator mechanism

- Random seed : a two-dimensional random value (a normal distribution)
- Dense : a fully connected layer
- Sigmoid : determined as 0 or 1
- RBS : $n \cdot k$ bit stream

Outline

1. Introduction
2. Background
3. Proposed Method
   - Design of Generator Model
   - Design of Predictor Model
   - Design of GAN based PRNG
4. Evaluation
5. Conclusion

Split the Generator's output

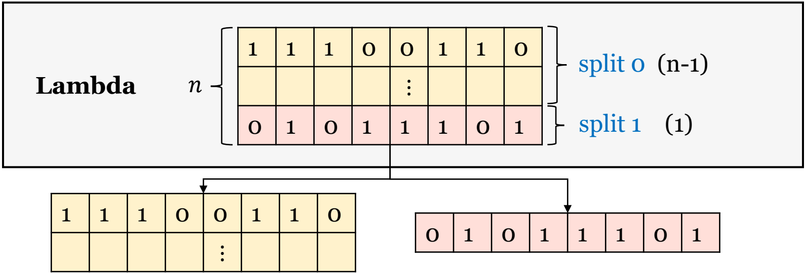

Figure: Split the Generator's output into 2 parts.

Split the Generator's output to input into the Predictor:

- $Split0$: Use as input to Predictor for training.
- $Split1$: Compared with Predictor's output.

Architecture of Predictor

RNN layer trains time series data:

- Enter $Split0$ as input
- Train $(n-1) \cdot k$-bit stream
- Predict k-bit stream

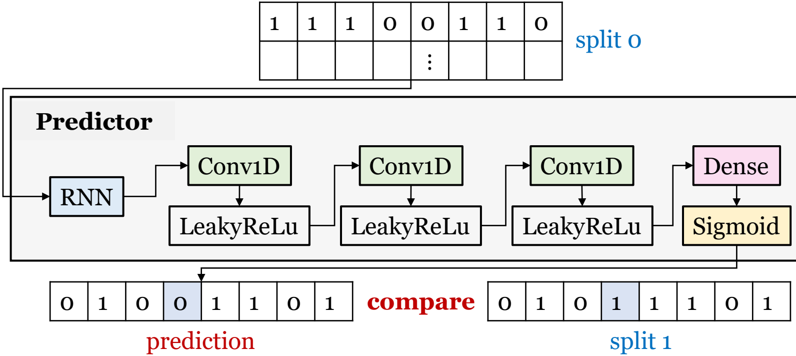

Figure: Architecture of Predictor.

Architecture of Predictor

Recurrent Neural Network (RNN) trains a bit stream:

- Each data has 8 features.
- Long-term dependence → Train about longer sequences.
- Can predict data following random walk.
- Prediction by considering the features of data in the desired range.

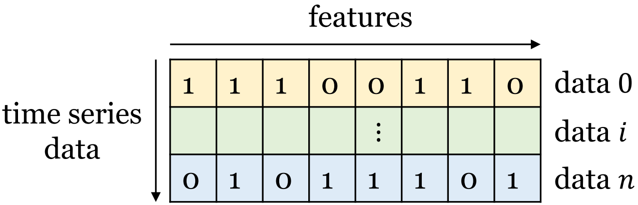

Figure: The Recurrent Neural Network (RNN) layer.

Architecture of Predictor

Algorithm 2 Predictor mechanism

Input: Random bit stream (RBS)
Output: Predicted random bit stream (RBS_P)

$Split0 \leftarrow RBS[: n-1][: 8]$
$Split1 \leftarrow RBS[n-1 : n][: 8]$
$x \leftarrow RNN(Split0)$
for $i = 1,$ **to** 3 **do**
    $x \leftarrow Conv1D(x)$
end for
$x \leftarrow Dense(x)$
$x \leftarrow Sigmoid(x)$
$RBS_P \leftarrow$ round x into nearest integer (0 or 1)
$Loss_p \leftarrow mean(abs(Split1 - RBS_P))$
Train to minimize $Loss_p$
return $RBS_P, Split1$

Figure: Predictor mechanism

- If $RBS_P == Split1$: RBS is predictable and $Loss_P$ is minimum
- Train to minimize $Loss_P$ for a correct prediction

Outline

GAN based Pseudo Random Number Generator

Generator is trained by combined model (Generator and Predictor):

- Reflecting the learning result of the Predictor.
- By adding an RNN layer to the Predictor, the overall performance is improved.

It's trained to generate unpredictable data, so it's very rare for the same pattern to occur periodically. Thus, a random bit stream is generated.

GAN based Pseudo Random Number Generator

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)

for $epoch = 1$ **to** $EPOCHS$ **do**
  $s \leftarrow$ sample $entropy$ from IoT device
  $RBS \leftarrow G(s)$
  $RBS_P, Split1 \leftarrow P(RBS)$
  $Loss_G \leftarrow mean(abs(1 - Split1 - RBS_P)) \cdot 0.5$
  Train G to minimize $Loss_G$
  $RBS \leftarrow G(s)$
end for
$c \leftarrow \sum_{i=0}^{m+t-1} 2^i \cdot RBS_i$
$num \leftarrow c \bmod r$
return num

Figure: GAN based PRNG mechanism

- If $RBS_P \; != Split1$: RBS is unpredictable and $Loss_G$ is minimum.
- Train to minimize $Loss_G$ for generation of RBS.

GAN based PRNG for Embedded Processors

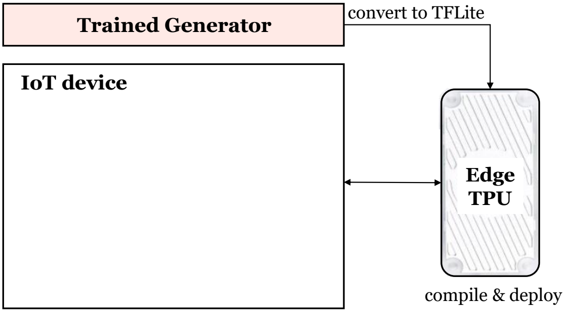

Figure: Trained Generator deployed on Edge TPU for embedded processors.

Only the Generator is deployed to the Edge TPU for embedded processor.

- It is a Generator that generates a random bit stream (Not a Predictor).
- Implemented with a simple architecture for limited environments.

GAN based PRNG for Embedded Processors

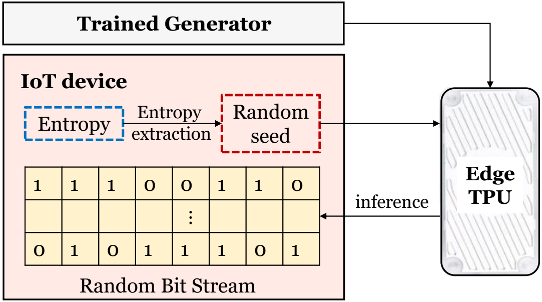

Figure: Generate a random bit stream in an embedded processor using Edge TPU

The entropy is collected on embedded processors as random seed (e.g. sensor data).

- The trained Generator has fixed weights, so a secure entropy is required.

Outline

Visualization of random number generated by the generator

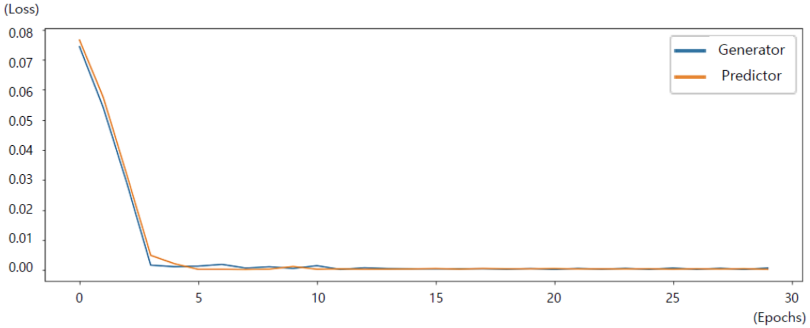

Figure: Loss of Generator and Predictor.

Both Generator and Predictor were trained to minimize loss.

- The Predictor can predict the Generator's output.
- The Generator generates a random bit stream that the Predictor cannot predict.

Visualization of random number generated by the generator

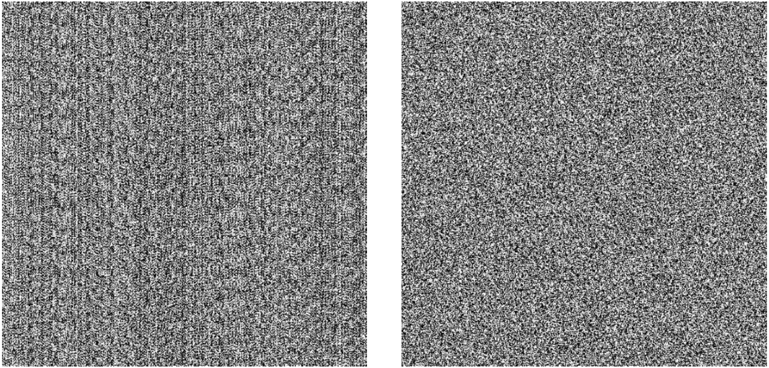

Figure: Visualization of random number generated by the generator. (left) before training and (right) after training.

Changes in internal state during training

$\rightarrow$ The randomness is improved.

Comparison with related work

Table: Comparison of data type, ratio of seed to output and mini-batch

	Output type	Seed : Output (bits)	Mini-batch	Output/Epoch
Bernardi et al. [2]	Decimal	64 : 262,144	400	104,857,600
This work	Bit	64 : 1,099,200	100	109,920,000

- For random seeds of the same length, more than four times the bits are learned at a time.
- This work achieves a similar level of randomness, up to 2.5 million bits per mini-batch.

NIST SP 800-22 : Randomness test for PRNG

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

Figure: Final Analysis Report of NIST test suite; (Left side) Bernardi et al. [2], (Right side) proposed method.

- Previous work mainly failed in Frequency, CumulativeSums, Run, FFT. → **Overcome by using time series neural networks (RNN).**
- This work failed the NonOverlappingTemplate once for the entire individual test.

NIST SP 800-22 : Randomness test for PRNG

Improve the randomness and overcome the problems of predictability and reproducibility of the previous method.

Table: 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. [2]	188	1830	56	3.0	2.7	4.5	2.5	187.09 ms
Proposed method (1)	188	1820	20.1	1.1	0.3	0.1	0.16	196.41 ms
Proposed method (2)	188	1794	19.6	1.09	0.00	0.1	0.00	13.27 ms

- P-value : failed 3 in (1), all passed in (2) and 27 in the previous method.
- Individual tests : reduced by about 45 times compared to the previous work.
- Inference time : $14 \sim 15$ times faster than the previous method on the desktop.

Outline

Conclusion and Future Work

- **Conclusion**
  - A novel GAN based PRNG for embedded processors.
  - Generation of hexadecimal and decimal numbers in variable lengths.
  - Porting lightweight GAN based PRNG to the Edge TPU.
  - High randomness validation through the NIST test suite.
- **Future Work**
  - Applying other GAN models for high randomness and efficiency.
  - Reducing the random seed length for resource-constrained environment.

Thanks and Questions

Thanks for your attention!

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)*, pages 265–283, 2016.

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*, pages 191–200. Springer, 2018.