




Lightweight Pseudo Random Number Generator for Embedded Systems

Andi Sama^{1*}, Meyliana², Yaya Heryadi¹, Taufik Roni Sahroni³

¹ Computer Science Department BINUS Graduate Program - Doctor of Computer Science, Bina Nusantara University, Jakarta 11480, Indonesia

² Information System Department School of Information System, Bina Nusantara University, Jakarta 11480, Indonesia

³ Industrial Engineering Department BINUS Graduate Program - Master of Industrial Engineering, Bina Nusantara University, Jakarta 11480, Indonesia

Corresponding Author Email: andi.sama@binus.ac.id

Copyright: ©2024 The authors. This article is published by IETA and is licensed under the CC BY 4.0 license (<http://creativecommons.org/licenses/by/4.0/>).

<https://doi.org/10.18280/ijse.140409>

ABSTRACT

Received: 11 May 2024

Revised: 16 July 2024

Accepted: 1 August 2024

Available online: 30 August 2024

Keywords:

embedded system, lightweight, PLC, PRNG, programmable logic controller, pseudo-random number generator, runs-test, simulation

A cryptography algorithm for data transfer encryption provides confidentiality, requires considerable computing power, and is not commonly implemented in embedded systems with limited computing power, such as Programmable Logic Controller (PLC). PLC is the core component for automation and control in industrial automation. For decades, PLC has prioritized speed over security; program execution in PLC must be as efficient as possible. The cryptography algorithm uses a seed, the initialization vector, to encrypt the data with the cryptography key to strengthen the encryption. Pseudo Random Number Generator (PRNG) can be used as the initialization vector. This paper proposes the XORasm PRNG algorithm, the lightweight XORshift-based algorithm with a modified seed from the system's clock. The applied methodology generates and visualizes PRNG, tests the randomness, and implements the PRNG on compact PLC. XORasm is evaluated statistically with runs-test in simulation by comparing the algorithm to one of the simulated compact PLC's PRNG implementations. The findings from this research are that p-values demonstrate that XORasm is statistically and significantly more random than the current implementation, and there is evidence that XORasm's generated data distribution is practically random at a 99.95% confidence level, suitable for implementation in embedded systems as a lightweight PRNG.

1. INTRODUCTION

In Industrial Automation, Manufacturers of embedded systems like PLC design simple and very efficient algorithms [1]. Automation and control have existed for decades [2]. One of the earliest industrial automation papers was in "Cybernetics: or Control and Communication in the Animal and Machine" by Wiener, N. in 1948 [3]. There are four eras: 1. early control (before 1900), 2. pre-classical (1900-1940), 3. classical (1935-1960), and 4. modern control (1995-now). The Industrial Revolution was referenced in 1776 [4].

Since the 1970s, PLC [5, 6], the core automation component in industrial automation [7], has been the catalyst for revolutionizing the industrial transition from mechanical (Industry 2.0) to computerized (Industry 3.0) by replacing mechanical timers, counters, and relays with programmable equivalents.

As part of the embedded systems (a set of hardware and software with limited computing power and resources designed for a specific purpose [8]), there are some identified issues in PLC security [9]. As PLC prioritizes speed over security, for many years, security has been a significant concern [10]. Typical attacks exploit the PLC-based system's vulnerabilities in the communication industry protocols. The payload attack is a key concern against data integrity in

Industry 4.0, particularly in IT/OT integration [11]. Encryption is part of the confidentiality of cryptography, which protects data at rest or in transit. Without encryption, data is stored or transferred in plaintext [12, 13].

Cryptography algorithms depend on an initialization vector, a fixed value, or randomly generated by RNG for increased security [14]. The lack of implementation of RNG in embedded systems such as PLC makes it challenging to perform security-related functions, such as data encryption. The initialization vector is typically used as the seed for modern cryptography algorithms, AES, for example, the accepted standard of symmetric encryption [15, 16], widely adopted since weaknesses found in DES and 3DES [17].

Existing PLCs mostly run at 32 bits—thus, the addressing capacity is up to 2^{32} (4,294,967,296) locations. A 64-bit PLC with an addressing capacity of up to 2^{64} locations is less likely to be needed. Older PLCs run at 8 or 16 bits.

RNG is commonly implemented as PRNG (usually software-based) [18, 19] rather than TRNG [20, 21], usually hardware-based [22-25]. PRNG is a trivial function in high-level programming languages (e.g., Java or Python). There is also a hardware implementation [7]. In PLC, RNG is not usually part of the standard functionalities in the PLC's common programming languages: LAD/LLD, ST/STX, SFC, or FB.

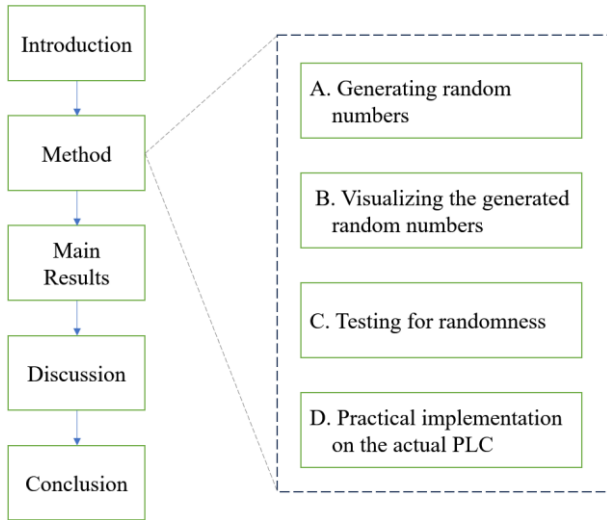


Figure 1. Paper organization

Stronger PRNG is possible (64 bits and above) for PC or PC Servers based on the XORshift algorithm (fast and lightweight) [26] and further explored in previous studies [27, 28]. However, PLCs, as embedded systems, commonly operate in 32 bits. While PRNG has been implemented using the system timer, the PRNG algorithm for generating better randomness for embedded systems such as PLCs is underexplored.

The research questions in this paper are RQ1: “Can we have a better PRNG algorithm for lightweight embedded systems?” RQ2: What is the base for measuring such an algorithm, and how can we measure such an algorithm for better randomness?” and RQ3: How can we apply the algorithm to embedded systems?”

The key theoretical contribution in this paper is the XORasm PRNG algorithm, the PRNG-type algorithm based on the XORshift algorithm, targeted for lightweight, low-power computing systems with generated random numbers that are statistically proven and randomly distributed at a 99.95% significance level. The main practical contribution is that XORasm can be used as the PRNG for lightweight, low-power computing power embedded systems. This is demonstrated through experiments on the actual PLC.

Figure 1 illustrates the paper organization. It starts with the introduction, followed by the method and main results, and ends with the discussion and conclusion sections. The method

consists of four stages: generating random numbers, visualizing the generated random numbers, testing for randomness, and practical implementation on the actual PLC.

2. METHOD

Figure 2 illustrates the four stages to generate two different sets of random numbers. A: Generate two sets of random numbers with XORasm and LGF algorithms, B: Visualize the generated random numbers, C: Test with runs-test in dieharder testing suite [29] to produce p-values and analyze the randomness using p-values, and D: Implement the XORasm on an embedded system (using FB, LAD, and ST programming language on a PLC). One implementation of RNG in PLC is in Siemens’ LGF.

2.1 Stage A – Random number generation

First, 32-bit random numbers are generated with XORasm—simulated in a laptop rather than in PLC due to constrained resources. Second, 32-bit random numbers are also generated with a simulated Siemens LGF random function in the same laptop. In this stage within Figure 1, XORasm and simulated LGF random functions generate 2 to the power of 32 random numbers. This is written as pow (2, 32) in C language. The pow (base, exponent) is a function as part of the math library. With 4 Bytes for each random number (32 bits equals 4 Bytes), the 2^{32} random numbers stored in disk space are 4 Bytes times 4,294,967,296, which equals 17,179,869,184 Bytes (16GB). Therefore, two datasets of four billion random numbers, with a size of 4 Bytes for each random number, totaling 16GB per file, are generated. Those two datasets are treated as two different independent distributions.

Figure 3 illustrates the pseudocode for generating seed for XORshift. On the right is the pseudocode of XORshift (run multiple times). In combination, the code is called XORasm. The seed is generated by computing the nanosecond portion of system time and storing it in 32 bits as unsigned integers. XOR operation with $\log_2(2^{32})$ is performed on each Byte of the system time. For the 32 bits, the system time is XOR-ed with the constant 0x05050505 (0x represents the following values in hexadecimal), which enhances randomness. The seed, representing the initial value of the XORshift algorithm, is stored in a file for reproducibility.

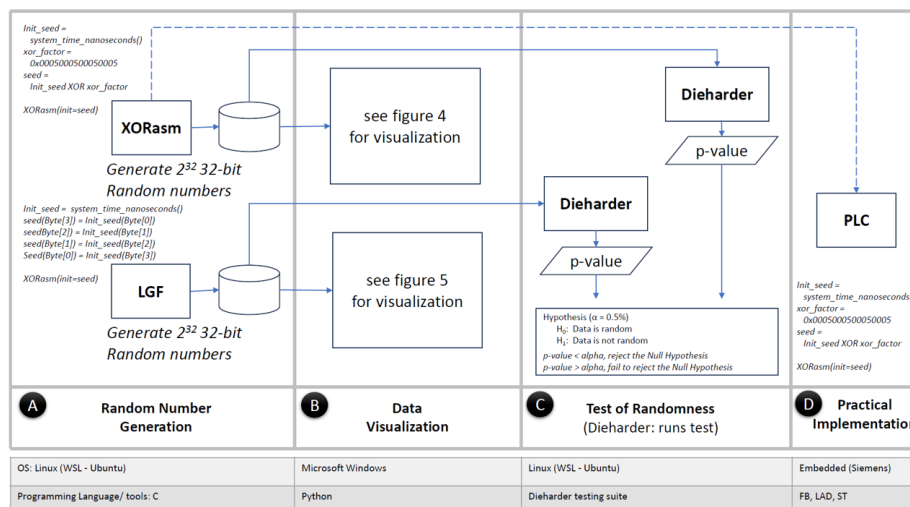


Figure 2. Method for generating, visualizing, and testing random numbers

```

Initialization: // plc_addressing = 32 for 32 bits
time_ns = system_time_nanoseconds();
Byte0 = xor_factor_base = log2(plc_addressing)
Process: // there are 4 Bytes for 32 bits
for i=3 to 1, i--b
Byte[i] = xor_factor_base << (8*i)
xor_factor = Byte3 ^ Byte2 ^ Byte1 ^ Byte0 // combine all
Output: // seed is 32 bits unsigned integer; the initial input to the XORshift
// time_ns XOR with xor_factor
seed = time_ns ^ xor_factor

// XORshift algorithm, with select shifts: 13, 17, 5
// Bit operations (like in C language):
// ^= is XOR with assignment
// << is shift left
// >> is shift right
static uint32_t xorshift32() {
seed ^= seed << 13;
seed ^= seed >> 17;
seed ^= seed << 5;
return seed;}

```

Figure 3. Xorasm pseudocode

2.2 Stage B - Visualizing the generated random numbers

Figure 4 illustrates the normalized distribution of the first 10,000 generated random numbers for the XORasm. Figure 5 illustrates the normalized distribution of the first 10,000 generated random numbers for the simulated LGF random function. Both are used by dieharder for testing. Please refer to stage B in Figure 2.

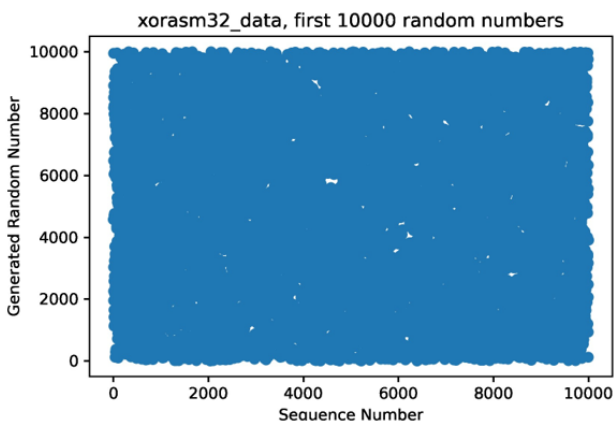


Figure 4. Normalized distribution (XORasm)

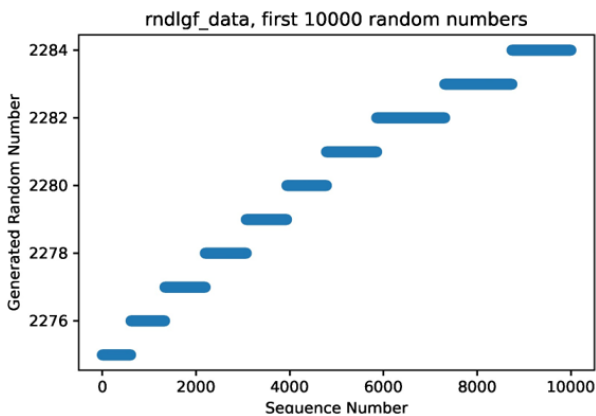


Figure 5. Normalized distribution (simulated LGF)

Normalized distribution means the chart does not directly visualize the random numbers, as the 32 bits of generated random numbers are too big for the chart to handle. Using Python programming, the generated random numbers are first normalized to be within the 0.0 to 1.0 (0-100%) range by dividing the random numbers by $2^{32} - 1$, the highest random numbers produced in 32 bits. This is step one. The result is then mapped to a certain range for plotting purposes and, in this case, mapped to minimum = 0 to maximum = 10,000. The normalized generated random numbers for visualization are minimum + results_from_step_one * (maximum - minimum).

Therefore, $0 + \text{results_from_step_one} * (10,000 - 0)$ becomes $\text{results_from_step_one} * 10,000$. This is for the y-axis. The x-axis is a sequence of 32 bits of unsigned integer numbers ranging from 0 to 10,000.

2.3 Stage C - Testing for randomness and technique for statistical randomness testing

2.3.1 Testing for randomness

Dieharder tests the data generated from the XORasm and the simulated LGF random function using a custom-developed Linux shell script for each sample size. Each time it executes, it performs two runs-tests. Thus, for five runs-tests for each sample size, the script generates ten p-values and ten assessments. Please refer to stage C in Figure 2. The pseudocode for the LGF random function is illustrated below. First, the function exchanges Byte 3 and 0 of the generated system's time (nanosecond part). Then, Byte 2 and Byte 1 are swapped.

The dieharder processes the generated random numbers. The following options were used to run the dieharder, selecting only necessary tests. “-d test_number” is set to run only the intended test for randomness. “-t sample_size” is set for taking the sample_size from the filename, how many random numbers that we want to test. “-p psample” is set for how many times we want to run the test to generate the p-value. “-f filename” is the file name containing the binary sequence of 32 bits generated random numbers, 4 Bytes each (\$dieharder -d test_number -t sample_size -p psample -f filename).

For the 32-bit XORasm algorithm, XORasm32 algorithm, the following command runs dieharder from the Linux shell:

```
“$dieharder -d 15 -t 10000 -p 100 -f xorasm32_data.bin.”
```

Options are set: “-d 15 option” to run only the runs-test for randomness, “-t 10000” to generate ten thousand random numbers with 4 Bytes each (32 bits). “-f filename” is the file name containing the binary sequence of 32 bits numbers (4 Bytes each), in this case, filename: *xorasm32_data.bin*.

2.3.2 Techniques for statistical randomness testing

The p-values output from dieharder is used for statistical tests. The p-value for testing randomness has the following conditions: Null Hypothesis (H_0) and Alternate Hypothesis (H_1). H_0 = Data is random. H_1 = Data is not random.

By using alpha = 0.005 (0.5% confidence level) as set by default in dieharder, the decision will be as follows (equivalent to the assessment in the dieharder test result):

If p-value \leq alpha, then we reject the Null Hypothesis. At (1-alpha) confidence level, we believe the evidence shows that the data is statistically significant and not random. There is a chance, at p-value, that the data is random.

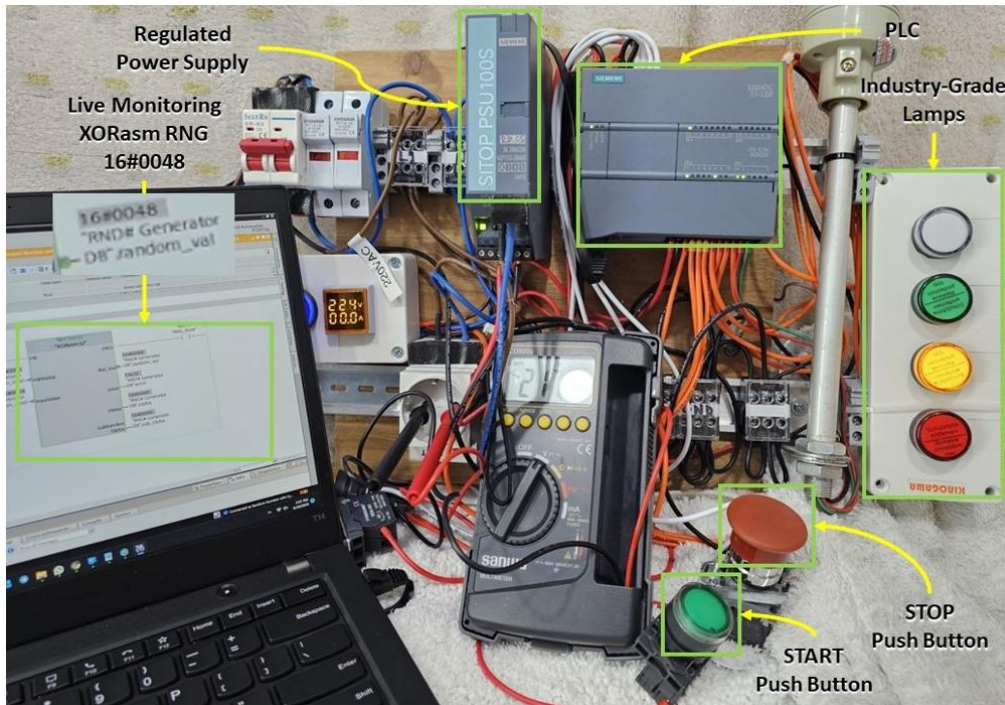


Figure 6. Experiment setup

If $p\text{-value} > \alpha$, we fail to reject the Null Hypothesis. At $(1-\alpha)$ confidence level, we believe the evidence shows that the data is random (statistically insignificant).

2.4 Stage D - Practical implementation on the embedded system, a PLC

We apply XORasm to an embedded system, the Siemens 1200 Compact PLC. SCL includes a callable function that implements the XORasm. The initial seed (32-bit unsigned doubleword) is derived from the nanoseconds portion of the PLC's system time. FB is called from the LLD program, which resides within the main program block. The main program transmits a signal to industrial-grade lamps, switching ON or OFF randomly.

Figure 6 illustrates the experiment setup in which a Siemens Simatic S7-1200 PLC runs the compiled program driving four industrial-grade lamps: White, green, yellow, and red. The industrial-grade lamps act as the actuator and are configured to connect to the PLC's digital outputs, the PLC DO ports. In the illustration, the yellow lamp is in the ON state, as the current value from the program output, as shown on the laptop's monitor, is x48 (in hexadecimal, translated to 72 in decimal). The output 72 is between 51 and 75, including 51 and 75, in which the programmed condition triggers the yellow lamp to an ON state. When a lamp is in the ON state, all other lamps are set to the OFF state.

Figure 7 illustrates the pseudocode of the simulated LGF random function.

A 24VDC power supply powered the PLC, the Siemens SITOP PSU 100S, which sources power from the regular 220VAC. In this setup, the voltages for all digital inputs and outputs are 24VDC.

Table 1 lists the values range and the corresponding lamps the program switches ON. Figures 8 and 9 illustrate the pseudocode of the XORasm algorithm (generating the initial seed and the main function, respectively), and Figure 10 illustrates the pseudocode of the PLC main program.

```
// Simulated Siemens LGF Random function
uint32_t rndlgf(uint32_t *seed) {
// Siemens PLC's implemented random function in
// LGF (library of general functions)
// The time part of the nanosecond (seed) is modified
// by simply switching "Byte 3 and Byte 0" and "Byte 1
// and 2" of the seed
uint32_t x = *seed;
uint32_t new_seed = 0, Byte3 = 0, Byte2 = 0, Byte1 = 0,
Byte0 = 0;
// switch Byte 0 and 3, Byte 1 and 2 of the seed
Byte3 = (x & 0x000000FF) << 24; // Byte 0 to Byte 3
Byte2 = (x & 0x0000FF00) << 8; // Byte 1 to Byte 2
Byte1 = (x & 0x00FF0000) >> 8; // Byte 2 to Byte 1
Byte0 = (x & 0xFF000000) >> 24; // Byte 3 to Byte 0
new_seed = Byte3 & Byte2 & Byte1 & Byte0;
*seed = new_seed;
return x; // return the switched Bytes}

```

Figure 7. Pseudocode (simulated LGF random function)

Table 1. Values range and lamps to switch ON

	Industrial-grade Lamps			
	White	Green	Yellow	Red
0-25	ON	OFF	OFF	OFF
26-50	OFF	ON	OFF	OFF
51-75	OFF	OFF	ON	OFF
76-100	OFF	OFF	OFF	ON

The laptop configuration includes an Intel i7 with 2.8GHz and 4 Cores processors, 16GB RAM, 1TB SSD storage, a Windows 11 OS, and Python version 3.19 with Anaconda. It also includes WSL2 with Ubuntu v22.00.04, a gcc compiler version 11.3.0, and the dieharder RNG testing suite version 3.31.1.

The source program was developed using the Siemens TIA Portal software application, version 17. The source program was compiled and downloaded to the PLC through an ethernet cable connecting the laptop to the PLC directly without an ethernet hub.

The PLC program starts when the green push button is pressed and stops when the red push button is pressed. The green and red push buttons are set as sensors and connected to the PLC's digital inputs (DI ports).

```
// System Time as initial seed
// get system time, the nanoseconds part that changes very fast
uint32_t time_ns = get_system_time_nanoseconds();
// xor_factor, a constant based on log2(plc_addressing)
// to make it more random. The base2 log on the plc
// addressing. now 32 bits served as the base for xor_factor.
// to be xor-ed with the system time just to make the seed
// slightly more random
uint32_t plc_addressing = 32; // 32 for 32 bits
uint32_t xor_factor_base = log2(plc_addressing);

// shift left xor_factor_base 24 bits for the MSB part, shift left
// xor_factor_base 16 bits, shift left xor_factor_base 8 bits,
// shift left xor_factor_base 0 bits for the LSB part, then
// combine all the 32 bits
uint32_t Byte3 = xor_factor_base << 24;
uint32_t Byte2 = xor_factor_base << 16;
uint32_t Byte1 = xor_factor_base << 8;
uint32_t Byte0 = xor_factor_base;
uint32_t xor_factor = Byte3 ^ Byte2 ^ Byte1 ^ Byte0;

// Modified seed, xor-ed with the constant, as the final seed to
// XORshift algorithm; set the seed to the XORshift algorithm
// by xor-ing the original time_ns with the xor_factor
uint32_t seed = time_ns ^ xor_factor;
```

Figure 8. Pseudocode (XORasm) – Inital seed

```
// XORasm32 Initialization function
// global variable: justRestarted (Boolean) is set to TRUE
// at System Restart; initialSeed is 32 bits unsigned variable
XORasm32_Init() {
    if justRestarted {
        initialSeed = System_Time_Nano_Seconds_Part
        // Initial seed is from System Clock, Nanoseconds part
        write(initialSeed) to PLC_Data_Block; }
// XORasm32 Algorithm function
// tempVar, initialSeed, seed are 32 bits unsigned variables
// xor_factor is 32 bits constant, log(pow(2,5)) = 5
// then the results are distributed per Byte
// xor_factor (in hexadecimal) = 0x0005000500050005
// normalized_random is 32 bits real number
// max_32bits is constant, pow(2,32) - 1
XORasm32(min, max) {
    if justRestarted {
        Initial_seed = Read(initialSeed) from PLC_Data_Block
        // Increase Randomness, seed XOR-ed with xor_factor
        seed = XOR(Initial_seed, xor_factor)
        write(seed) to PLC_Data_Block
        justRestarted = FALSE
    } else {
        // XORshift algorithm, then normalize to 0-1
        seed = XOR(tempVar, SHL(tempVar, 13))
        seed = XOR(tempVar, SHR(tempVar, 17))
        seed = XOR(tempVar, SHL(tempVar, 5)) }
    normalized_random = seed / max_32bits;
    // Scale according to min to max
    return realToInteger(min + (max-min)*seed); }
```

Figure 9. Pseudocode (XORasm) – Main function

Initially, all lamps are set to the OFF state by setting each of the DO ports connected to them to 0VDC (OFF state). To

set the lamp to the ON state, the corresponding DO port is set to 24VDC. When the PLC program runs, Siemens TIA Portal software shows live real-time monitoring of real-time values. The TIA Portal Software is the digital twin of the real PLC hardware running the program.

```
// PLC main program
// when the start push_button pressed loop continuously
// calling the XORasm, activate the selected lamp
// when the stop push_button pressed, stop the loop until
// start push_button is pressed again
main() {
    init();
    while StartPushButtonPRESSED { do {
        case normalize(XORasm(minimum, maximum)):
            0..25 PLCDOPort(redLampPort) = ON;
            26..50 PLCDOPort(yellowLampPort) = ON;
            51..75 PLCDOPort(greenLampPort) = ON;
            76..100 PLCDOPort(whiteLampPort) = OFF;}}
        delay(delayTime); clearAllLamps();
    } while StopPushButtonPRESSED; }
// Supporting functions
// set the min and max values to normalize the output of
// XORasm to be only between 0-100. setting the physical
// ports of the PLC digital outputs connected to the lamps
init() {
    minimum=0; maximum=100;
    delayTime= 1000 // 1000 miliseconds
    redLampPort=PLCDOPort0; yellowLampPort=PLCDOPort1;
    greenLampPort=PLCDOPort2; whiteLampPort=PLCDOPort3;
    clearAllLamps(); }
clearAllLamps() { // set all lamps to OFF states
    PLCDOPort(redLampPort), PLCDOPort(yellowLampPort),
    PLCDOPort(greenLampPort), PLCDOPort(whiteLampPort)
    = OFF; }
```

Figure 10. Pseudocode (PLC program)

3. MAIN RESULTS

Table 2 summarizes the statistical analysis for both algorithms: The XORasm and LGF random function. The table lists the p-values of the XORasm runs test's simulation result compared to the p-values of the simulated LGF random function for a sample size of 10,000 (dieharder's default). The assessment is PASS if the p-value > alpha (p-value above 0.5%) and WEAK if the p-value <= alpha.

The data distribution is random if the p-value has a value above alpha. All p-values are consistently above alpha. Therefore, all p-values are statistically non-significant, meaning we fail to reject the Null Hypothesis.

For XORasm, all generated data are tested through runs test with p-values above alpha. Therefore, we believe the evidence shows that the XORasm's data distribution is random (statistically insignificant).

In contrast, for the simulated LGF random function, the p-values have values below alpha several times. Three tests with p-values 0.0001234, 0.00026679, and 0.00320486 are statistically significant, meaning we reject the Null Hypothesis for those p-values. We accept the alternative hypothesis (data is not random for all p-values below alpha). Therefore, we believe the evidence shows that the data distribution for the LGF random function is statistically significant and not random. There is a chance, at p-value, that the data is random.

Table 2. XORasm runs-test and LGF random function

Dieharder Testing Suite for Randomness (runs test) ^{*2)}											
Runs Test ^{*3)} for sample size: 10,000 ^{*1)}											
	1		2		3		4		5		
	1	2	1	2	1	2	1	2	1	2	
A. XORasm algorithm											
- p-value	0.1086568	0.0490162	0.1226328	0.0067232	0.2227413	0.2219964	0.0298953	0.0546924	0.0931555	0.2600096	
- assessment	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED
B. Siemens PLC random algorithm ^{*4)}											
- p-value	0.0001234	0.7644705	0.0002668	0.0032049	0.1014579	0.7473626	0.0161754	0.1249053	0.0533751	0.0919376	
- assessment	WEAK	PASSED	WEAK	WEAK	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED	PASSED

^{*1)} dieharder default sample size.

^{*2)} The filename for xorasm algorithm: "xorasm32_data.bin", and for simulated Siemens' PLC: "rndlgf_data.bin".
The random numbers are taken from a .bin file. Number of generated random numbers in each .bin file: 4,294,967,295 Bytes.
Each random number is 4 Bytes (32 bits). Thus, total file size is 16GB (Giga Bytes).

^{*3)} Simulated and tested by the dieharder RNG testing suite on an x86 laptop.

^{*4)} The Siemens PLC LGF random algorithm function simulated on an x86 laptop.

4. DISCUSSION

Based on the p-value results, statistically, we fail to reject the Null Hypothesis. We conclude that there is evidence that the data distribution generated by the XORasm PRNG algorithm is practically random at a 99.95% confidence level (Table 2).

In contrast, from the p-values results for the LGF random function runs-test, statistically, we conclude that there is evidence that data distribution generated from the LGF random function algorithm is not random at a 99.95% confidence level. Random function in Siemens PLC LGF represents the design and implementation of common random number algorithm in embedded systems. Provided the hardware supports it, a better random generator to increase randomness can use another system clock or independent timing cycle [30].

The assessment results conclude that for the default sample_size = 10,000 sets by the dieharder RNG testing suite, the XORasm has a better random number distribution as it passed all the tests (assessment = PASSED). In contrast, the simulated LGF random function has the result, assessment = WEAK, in three of the ten runs, indicating it does not have a good distribution of random numbers for all the tests.

The findings show that XORasm is a good PRNG, according to runs-test performed in a dieharder testing suite.

The theoretical implication of this research is the contribution to the literature in Computer Science and Computer Engineering for embedded systems with lightweight, low computing power. The advantage for embedded devices is straightforward, as the XORasm PRNG algorithm is statistically proven and randomly distributed at a 99.95% significance level. While the algorithm is lightweight and suitable for embedded devices such as PLC, IoT, and edge computing devices, it provides better randomness over the random generator based on the system's clock. XORasm can serve as the seed for modern encryption algorithms such as AES. Possible further research is by setting a lower alpha value and inspecting whether the test conclusion still holds.

The practical implication is that the XORasm PRNG algorithm can be used for low-power computing devices such as PLC to increase data transfer security among embedded systems. Further research is possible using 64-bit PLCs or combining embedded systems with higher-power devices, including PCs and servers.

5. CONCLUSION

This research paper proposes a lightweight, software-based PRNG called XORasm for embedded systems such as PLC, based on the XORshift algorithm, combined with a modified seed taken randomly from the PLC's system clock. The randomness of the 32-bit XORasm PRNG algorithm is demonstrated through a simulation using statistics (runs-test) using the dieharder RNG testing suite. The result is statistically analyzed to the simulated Siemens implemented 32-bit random function, LGF_RandomRange_Dint, as part of Siemens' LGF. XORasm demonstrated in Siemens 1200 Compact PLC. This has answered RQ1: Yes, and RQ2: Taking one of the implementations in PLC as a base, the LGF in Siemens PLC, and using statistical runs-tests within the dieharder testing suite.

Based on the p-values analysis, the findings demonstrated that XORasm is statistically and significantly better at generating randomness than the current implementation. Statistically, at a 99.95% confidence level, there is evidence that the generated data distribution is practically random and suitable for implementation as a lightweight PRNG for embedded systems.

XORasm uses the XORshift algorithm with the seed, the initialization vector. XORasm significantly improves the randomness of generated random numbers and is suitable for lightweight embedded systems like PLC. By comparing XORasm and the simulated Siemens PLC implementation, the LGF random function, the XORasm implementation in Siemens PLC, has answered RQ3.

The experiment shows that the 32-bit version of the proposed XORasm successfully passed the dieharder RNG testing suite for runs-test. XORasm is generally suited to 32-bit PLCs and can be implemented using LLD, SFC, ST, or FB.

The XORasm PRNG algorithm (32 bits) is based on the XORshift algorithm. Its primary design purpose is to be fast and efficient for a lightweight system with limited computing power. The demonstrated implementation is limited to only one PLC, Siemens PLC.

This research's practical implications may include enhancing existing random number generation algorithms for PLCs, not just Siemens, and low computing power embedded systems in general, including edge computing devices.

Potential future work for theoretical contribution may include improving the algorithm, making the PRNG more random using a different modified seed, and testing using a

statistical runs-test with lower alpha (lower than 0.5%), considering lightweight computing power and limited resources in embedded systems.

Potential future work on the practical application could involve applying the XORasm PRNG algorithm to several 32-bit PLC brands, 16-bit earlier PLCs, CPS, and possibly 64-bit future PLCs. Using XORasm as a seed (initialization vector) for a symmetric cryptography algorithm (e.g., AES) in CBC mode, for example, can improve data transfer security between PLC and SCADA systems and other external systems such as MES or IoT.

REFERENCES

- [1] Tarnawski, J., Kudelka, P., Korzeniowski, M. (2022). Advanced control with PLC—Code generator for aMPC controller implementation and cooperation with external computational server for dealing with multidimensionality, constraints and LMI based robustness. *IEEE Access*, 10: 10597-10617. <https://doi.org/10.1109/ACCESS.2022.3142054>
- [2] Cheng, F.T. (2021). Evolution of automation and development strategy of intelligent manufacturing with zero defects. In: Cheng, F.T. (e.d.) *Industry 4.1: Intelligent Manufacturing with Zero Defects*. IEEE Press, New York, USA, pp. 1-23. <https://doi.org/10.1002/9781119739920.CH1>
- [3] Bennett, S. (1996). A brief history of automatic control. *IEEE Control Systems Magazine*, 16(3): 17-25. <https://doi.org/10.1109/37.506394>
- [4] Smith, A. (1997). *An Inquiry into the Nature and Causes of the Wealth of Nations*. University of Chicago Press, Chicago, USA. <https://doi.org/10.7208/chicago/9780226763750.001.0001>
- [5] Zaheer, M.A., Nauman, M., Fai, R.B. (2022). Identify components to manage security requirements in RAMI 4.0: An explanatory case study on industrial architecture. *Advances in Automobile Engineering*, 11(2): 1-7. <https://doi.org/10.35248/2167-7670.1000186>
- [6] Luo, J., Kang, M., Bisse, E., Veldink, M., Okunev, D., Kolb, S., Canedo, A. (2020). A quad-redundant PLC architecture for cyber-resilient industrial control systems. *IEEE Embedded Systems Letters*, 13(4): 218-221. <https://doi.org/10.1109/LES.2020.3011309>
- [7] Namekar, S.A., Yadav, R. (2020). Programmable Logic Controller (PLC) and its applications. *International Journal of Innovative Research in Technology (IJIRT)*, 6(11): 372-376.
- [8] Grycel, J.T., Walls, R.J. (2019). A random number generator built from repurposed hardware in embedded systems. *ArXiv*. <https://doi.org/10.48550/arXiv.1903.09365>
- [9] Wu, H., Geng, Y., Liu, K., Liu, W. (2019). Research on programmable logic controller security. *IOP Conference Series: Materials Science and Engineering*, 569(4): 042031. <https://doi.org/10.1088/1757-899X/569/4/042031>
- [10] Buchanan, S.S. (2022). *Cyber-attacks to industrial control systems since Stuxnet: A systematic review*. Ph.D. Dissertation, Capitol Technology University, South Laurel, Maryland, USA. <https://doi.org/10.5555/AAI29163646>
- [11] Wang, Z., Zhang, Y., Chen, Y., Liu, H., Wang, B., Wang, C. (2023). A survey on programmable logic controller vulnerabilities, attacks, detections, and forensics. *Processes*, 11(3): 918. <https://doi.org/10.3390/PR11030918>
- [12] NIST. (2024). *The NIST Cybersecurity Framework (CSF) 2.0*. <https://doi.org/10.6028/NIST.CSWP.29>
- [13] Stouffer, K., Pease, M., Tang, C.Y., Zimmerman, T., Pillitteri, V., Lightman, S., Hahn, A., Saravia, S., Sherule, A., Thompson, M. (2023). *Guide to operational technology (OT) security*. NIST Special Publication SP 800-82r3, Gaithersburg, Maryland. <https://doi.org/10.6028/NIST.SP.800-82r3>
- [14] Karell-Albo, J.A., Legón-Pérez, C.M., Madarro-Capó, E.J., Rojas, O., Sosa-Gómez, G. (2020). Measuring independence between statistical randomness tests by mutual information. *Entropy*, 22(7): 741. <https://doi.org/10.3390/E22070741>
- [15] Mameri, Z.Z. (2024). *Cryptography: Algorithms, Protocols, and Standards for Computer Security*. John Wiley & Sons, Inc., Hoboken, New Jersey, USA. <https://doi.org/10.1002/97811394207510>
- [16] Rastoceanu, F., Rughiniş, R., Tranca, D.C. (2023). Lightweight cryptographic secure random number generator for IoT devices. In *2023 24th International Conference on Control Systems and Computer Science (CSCS)*, Bucharest, Romania, pp. 180-185. <https://doi.org/10.1109/CSCS59211.2023.00036>
- [17] Dworkin, M.J., Barker, E., Nechvatal, J.R., Foti, J., Bassham, L.E., Roback, E., Dray Jr., J.F. (2023). *Advanced Encryption Standard (AES) (FIPS 197)*. Federal Information Processing Standards Publication, National Institute of Standards and Technology, Gaithersburg, Maryland. <https://doi.org/10.6028/NIST.FIPS.197-upd1>
- [18] Parisot, A., Bento, L.M.S., Machado, R.C.S. (2021). Testing and selecting lightweight pseudo-random number generators for IoT devices. In *2021 IEEE International Workshop on Metrology for Industry 4.0 & IoT (MetroInd4.0&IoT)*, Rome, Italy, pp. 715-720. <https://doi.org/10.1109/MetroInd4.0IoT51437.2021.9488454>
- [19] Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J., Vo, S. (2010). *A statistical test suite for random and pseudorandom number generators for cryptographic applications*. NIST Special Publication 800-22 Revision 1a, Gaithersburg, Maryland. <https://doi.org/10.6028/NIST.SP.800-22r1a>
- [20] Gomez, H., Arenas, J., Roa, E. (2020). Low-cost TRNG IPs. *IET Circuits, Devices & Systems*, 14(7): 942-946. <https://doi.org/10.1049/iet-cds.2019.0535>
- [21] Al-Shidaifat, A., Jayawickrama, C., Jung, Y., Song, H., Kahrama, N. (2020). Chaotic true random number generator for secure communication applications. In *2020 International SoC Design Conference (ISOCC)*, Yeosu, Korea (South), pp. 244-245. <https://doi.org/10.1109/ISOCC50952.2020.9333113>
- [22] Boyd, R. (1999). *Random number generator*. In: *Tolerance Analysis of Electronic Circuits Using MATHCAD*. CRC Press, Boca Raton, USA, p. 76. <https://doi.org/10.1201/9781315215402-27>
- [23] Nebhen, J. (2020). A low power CMOS variable true random number generator for LDPC decoders. In: *Goel*,

- N., Hasan, S., Kalachelvi, V. (eds) Modelling, Simulation and Intelligent Computing: Proceedings of MoSICom 2020. Lecture Notes in Electrical Engineering, vol 659. Springer, Singapore. https://doi.org/10.1007/978-981-15-4775-1_53
- [24] Jofre, M., Curty, M., Steinlechner, F., Anzolin, G., Torres, J.P., Mitchell, M.W., Pruneri, V. (2011). True random numbers from amplified quantum vacuum. *Optics Express*, 19(21): 20665-20672. <https://doi.org/10.1364/OE.19.020665>
- [25] Matsuoka, S., Ichikawa, S., Fujieda, N. (2021). A true random number generator that utilizes thermal noise in a programmable system-on-chip (PSoC). *International Journal of Circuit Theory and Applications*, 49(10): 3354-3367. <https://doi.org/10.1002/cta.3046>
- [26] Marsaglia, G. (2003). XORshift RNGs. *Journal of Statistical Software*, 8(14): 1-6. <https://doi.org/10.18637/jss.v008.i14>
- [27] Bhattacharjee, K., Das, S. (2022). A search for good pseudo-random number generators: Survey and empirical studies. *Computer Science Review*, 45: 100471. <https://doi.org/10.1016/J.COSREV.2022.100471>
- [28] Vigna, S. (2016). An experimental exploration of Marsaglia's XORshift generators, scrambled. *ACM Transactions on Mathematical Software (TOMS)*, 42(4): 1-23. <https://doi.org/10.1145/2845077>
- [29] Brown, R.G. (2023). Robert G. Brown's General Tools Page. Available: <https://webhome.phy.duke.edu/~rgb/General/dieharder.php>.
- [30] BIN95 (2023). A universal PLC programming example of how to generate a truer exponential random number. <https://bin95.com/articles/automation/plc-programming-random.htm/>.

NOMENCLATURE

3DES	Triple DES
AES	Advanced Encryption Standard
CBC	Cipher Block Chaining
CPS	Cyber-Physical Systems
DES	Data Encryption Standard
DI	Digital Input
DO	Digital Output
FB	Function Block
GB	Giga Bytes
GHZ	Giga Hertz
IoT	Internet of Things
IT	Information Technology
LAD/LLD	Ladder Logic Diagram
LGF	Library of Generic Functions (Siemens)
LSB	Least Significant Byte
MES	Manufacturing Execution System
MSB	Most Significant Byte
OS	Operating System
OT	Operational Technology
PC	Personal Computer
PLC	Programmable Logic Controller
PRNG	Pseudo-RNG
PSU 100S	Siemens Power Supply Unit, 100S-type
RAM	Random Access Memory
RNG	Random Number Generator
RQ	Research Question
SCADA	Supervisory Control and Data Acquisition
SFC	Sequential Function Chart
SSD	Solid State Drive
ST/STX	Structured Text
TB	Terabyte
TIA	Totally Integrated Automation
TRNG	True RNG
VAC	Voltage Alternating Current
VDC	Voltage Direct Current
WSL2	Windows Subsystem for Linux version 2
XOR	eXclusive OR bitwise operation