Comparative Analysis of CPU Jitter Across Technical Environments

Elizabeth Allen   
College of Business  
University of Alabama in HuntsvilleHuntsville, AL  
ea0073@uah.edu  
  
Austin Kelly   
College of Business  
University of Alabama in HuntsvilleHuntsville, AL  
ak0155@uah.edu  
  
Riddhi Patel   
College of Business  
University of Alabama in HuntsvilleHuntsville, AL  
rsp0015@uah.edu

Sriabhinay Kusuma   
College of Business  
University of Alabama in HuntsvilleHuntsville, AL  
sk0189@uah.edu

Robert Zecher   
College of Business  
University of Alabama in HuntsvilleHuntsville, AL  
rez0002@uah.edu

*Abstract*—This study performed an assessment of CPU Jitter version 3.6.1, as produced by Steven Mueller. Utilizing github code, provided by Mueller, research collected min-entropy measurements and unpredictability statistical measurements from four technical environments: bare metal with container, bare-metal, WSL container, and virtual machine. Min-entropy, in combination with unpredictability tests, provided by the National Institute of Science and Technology, were cross-analyzed to determine optimal environments for CPU Jitter usage and to ensure CPU Jitter version 3.6.1 meets cryptographic standards. Research found that bare-metal environments produced optimal min-entropy due to the closeness of the underlying hardware architecture, as compared to other environments. This research omits restart tests, health tests, and conditioning from its study, and only focuses the scope of study on bare-metal and containerized environments. This research is significant because it asserts that CPU Jitter version 3.6.1 meets industry standards for cryptography, but only in specific environments.

Keywords—entropy, environment, jitter, noise, predictor, random number generator

# Introduction

The National Institute of Science and Technology (NIST) has tested previous versions of CPU Jitter RNG to study if the design of the system is sufficient for industry-standard cryptography. CPU Jitter RNG is a non-physical random number generator, created by Steven Mueller [1]. It uses non-physical entropy sources derived from CPU execution timings. While previous versions of CPU Jitter RNG have been analyzed, the newest version, 3.6.1, has not. In this study, CPU Jitter version 3.6.1 will be tested in accordance with NIST Special Publication 800-90B: Recommendation for Entropy Sources Used for Random Bit Generation [2]. This publication acts the standard for assessing entropy sources, and it provides processes and tests to analyze if the entropy sources used by an RNG are sufficient for industry-standard cryptography. This study will also cross-analyze entropy derived from container, virtual machine, and bare-metal environments.

The significance of having an efficient benchmark system for CPU Jitter RNG is crucial in determining the quality of the entropy, which directly correlates to cryptographic security. Because entropy serves as the base of cryptographic security, it is essential to analyze if the entropy source feeding an RNG exhibits the level of unpredictability required for cryptographic use. If the entropy source does not meet requirements, then the RNGs using it will not provide strong, secure cryptographic keys, allowing openings for vulnerabilities and attacks.

# Literature Review

## Entropy Assessment

Entropy is a piece of quantifiable, unpredictable information that serves as a base for cryptographic security. Cryptographic systems heavily rely on the RNG for nonces, key generation, strong encryption schemes that require high-entropy keys, salting in hash functions, signature generation, and also in blockchain technologies. Entropy as an unpredictable factor is the foundation of security. Entropy sources must exhibit true randomness; otherwise, it could result in adversarial prediction.

Many modern operating systems derive entropy from hardware events like disk I/O, packet arrivals on the network, temperature factors, and more. Also, CPU-based entropy is derived from CPU execution timings. The real challenge arises in a virtual environment set up where the systems have no direct access to physical hardware events; CPU-based entropy systems address such cases [3] [4].

The system in which entropy is derived from the CPU execution timing is called a CPU Jitter. The tiny, unpredictable differences in the time taken for a CPU to run instructions are where the entropy is quantified. These differences depend on various factors, such as predefined processes in the system, such as the cache system, scheduling system, memory system, etc. Though these processes/systems are for their independent usage in the system, the quantifiable differences in instruction execution from these subsystems are not guessable.

Secure cryptographic algorithms depend on entropy from random number generators; without guidance from the industry standards that NIST provides, the availability and authenticity of these RNG options could out-scale and devalue cryptographic functions used throughout technology today. For a cryptographic method to be considered secure, it must incorporate random bit generation and avoid similar, consistent, and predictable entropy levels. NIST has provided SP 800-90B as a recommendation to test the entropy sources for RNGs [2].

The purpose of NIST SP 800-90B is to provide a standard recommendation for assessing the min-entropy exhibited by various entropy sources. Min-entropy measures the amount of unpredictability in a set of outcomes based on the probability of guessing the most likely output of a randomness source [2]. This recommendation prompts researchers to collect data from an entropy source and conduct a set of statistical tests to calculate an entropy estimate and then validate that the estimate meets industry standards to be used in RNGs. Larger entropy estimates indicate more randomness and unpredictability in the entropy source, while smaller entropy estimates may indicate a more predictable source

## SP800-90B NIID

According to NIST SP 800-90B, there are two test tracks used to assess entropy: Independently and Identically Distributed (IID) and Non-IID (NIID) [2]. The track chosen depends on the properties of the entropy data being examined. IID data is similar to flipping a coin or rolling dice. Each data point is independent from all other data points in the dataset, and each data point in an output sequence has the same probability of distribution. Essentially, a data point cannot be predicted by the point that comes before it and cannot be used to predict the point that comes after it. Data is considered to be NIID if it does not exhibit the properties of IID data, and it can be assumed that there is some level of dependency between data points. Testers can determine if data should follow the IID or NIID test track by proposing an IID assumption of the noise source that supports the entropy and by performing the IID track to test the assumption. If the entropy source fails the IID tests, the assumption cannot be statistically verified and will be considered NIID [2]. For this reason, previous studies on CPU Jitter have utilized the NIID track, and NIST SP 800-90B proposes seven tests:

* The Collision Test - This test was originally proposed by Entropy Bounds and Statistical Tests [5] to assess the randomness of a dataset. A collision is defined as repeated values in a dataset. This test estimates the mean number of values produced until a collision occurs. If a dataset is truly random, it will take longer for a collision to occur. Collisions will be more frequent if the dataset is less random, allowing some values to appear in the dataset more frequently.
* The Compression Test - This test was originally proposed by Entropy Bounds and Statistical Tests [5]. This test calculates the entropy rate of a dataset based on how much the data can be compressed.
* The Lag Test- The lag predictor contains several sub-predictors, each of which predicts the next output, based on a specified lag
* The Lz78y Test - The Lz78y predictor is an adaptation of LZ78 encoding, incorporating Bernstein's Yabba scheme for dictionary updates. It maintains a dictionary of previously added strings and continues expanding it until reaching maximum capacity. When processing a sample, all substrings from the most recent B samples either update or get added to the dictionary.
* The Markov Test - This test measures the dependencies between consecutive values in a dataset. This test also helps to determine randomness by determining if a past value impacts a future value. If the data presents the “Markov Process,” then the next data value only depends on previously observed data values. This test determines if the dataset follows the Markov Process or if there are other, unknown dependencies present. Unknown dependencies may impact the randomness of data.
* The Multi MCW Test - This assessment is compiled of multiple predictors, each aiming to guess a future output value based on a specified number of previous values (i.e., the window). Each predictor seeks to guess the number of times a value will appear in a window of values. The Multi MCW Test records the number of times each predictor was correct and then uses the predictor with the most correct guesses to predict the next value for the test as a whole.
* The Multi MMC Test - This assessment is compiled of multiple Marvok Models with counting predictors. Similar to the above Markov test, this assessment uses multiple predictors to guess future output values based on the previous output values. Each predictor runs in parallel, working to guess values simultaneously. The Multi MMC test records the number of times each predictor correctly guesses a future value. It uses the predictor with the most correct guesses to then predict the next value.
* T-Tuple Test - This assessment determines if a sequence of bits exhibits randomness by evaluating the occurrence of a median in in a binary data stream.
* LRS Test - This assessment tests for repetitive patterns and dependencies in bit streams that may indicate a lower level of unpredictability.

In the context of entropy assessment, especially when evaluating noise sources like CPU jitter for use in RNGs, predictor and non-predictor tests are two distinct approaches for analyzing randomness.

Predictor-based tests aim to estimate or forecast the next value in a sequence using previously observed data. These tests are designed to evaluate how well a model can learn patterns in the data and thereby quantify how unpredictable a source truly is. A good entropy source should resist prediction, meaning a low success rate in predictor-based estimations indicates high entropy. “Predictor tests... model the entropy source and attempt to guess the next output. They are instrumental in identifying overestimating entropy in sources that may exhibit some deterministic behavior”[5]. Common predictor tests include:

* LZ78Y compression test
* Markov models
* Multi-order prediction tests

Non-predictor tests, in contrast, assess statistical properties of a sequence without trying to guess future values. These include frequency analysis, autocorrelation, and distributional tests. They are used to detect non-random patterns or statistical anomalies in the observed data, but do not directly assess entropy or unpredictability. "Statistical tests... examine whether the observed data exhibit the expected properties of a random sequence, such as uniform distribution and lack of correlation"[5]. Examples include:

* T-Tuple
* LRS Tests
* Autocorrelation test
* Chi-square test

In jitter-based RNGs (such as those using jitterentropy-rngd), both types of tests are essential. While non-predictor tests may confirm statistical randomness, they can miss subtle patterns that a predictor might detect. Thus, passing predictor tests is a stronger indication of true entropy, especially in high-assurance cryptographic environments. Predictive models... are capable of capturing subtle dependencies in the source output that are missed by traditional statistical tests. Therefore, they are essential in assessing min-entropy under real-world operating conditions [5].

## Expansions and Limitations of 800-90B

The tests described above are standardized and can be performed on various entropy sources aside from CPU Jitter. For example, these same NIST 800-90B tests have been applied to other sources of entropy to assess if they are suitable for cryptographic use. One such source is signals derived from ECG readings, which are categorized as a physical entropy source rather than a virtualized one [6]. Another study is assessing non-identical inverter rings as an entropy source and is performing its assessment against NIST 800-90B [7]. A third study uses non-thermal electrochemical microplasma, which is essentially an electrically charged gas, as an entropy source [8]. All three studies have tested their entropy sources against the NIST 800-90B, and all three produced successful results according to the standards put in place. CPU execution timings, ECG signals, inverted rings, and microplasma are all vastly different sources of entropy, but each source can be assessed using NIST 800-90B tests. This wide range of entropy sources highlights the benefits of using the standardized test suites provided by NIST and shows that this is a reliable method of testing that can be used against many different entropy sources.

While the studies mentioned above focus on NIST 800-90B to design and test entropy sources, its processes can also be combined with two other NIST publications, NIST 800-90A and NIST 800-90C, to provide full instructions on creating an RNG that can be used for secure, unpredictable cryptography. An RNG requires an entropy source and an algorithm to convert entropy to produce a random output. NIST 800-90B focuses on designing entropy sources, and NIST 800-90A describes deterministic algorithms that take an entropy input and use it to produce pseudo-random values. NIST 800-90C provides the “glue” for putting the entropy source together with the algorithm to implement an RNG [9]. These publications provide a full roadmap to create an RNG. As shown by the wide range of possible entropy sources, this highlights that RNGs can be created and utilized using various sources paired with established algorithms. Future research could take CPU Jitter entropy sources, which have been tested with NIST 800-90B, combine the source with deterministic algorithms set forth in NIST 800-90A, and utilize NIST 800-90C to create a new RNG.

However, while it is shown that NIST 800-90B is utilized across industries, it is not without criticism. Some researchers have argued that tests in the NIID track may underestimate min-entropy [10]. This paper argues that the Collision Test and the Compression Test cannot reliably estimate entropy because both tests approximate entropy using a calculated statistic derived from the NIID data. The formula used to calculate the statistic assumes that the data actually has IID properties, which removes the assumption of dependencies within the data. The presence of dependencies is the key difference between IID and NIID, so these tests may assume less dependency within the test data, leading to an underestimation of entropy [10]. Another analysis on NIST 800-90B tests agreed with Zhu, et al., and found that the Collision and Compression tests underestimate entropy [2]. Both studies agree that future research could explore if the algorithms used in these tests can present vulnerabilities to attackers. Potential workarounds include restructuring the formula, so it does not assume IID characteristics. This solution is beyond the scope of this research, but it can be considered for future Jitter research.

In the context of Jitter 3.6.1 research, it is possible that the Collision and Compression tests performed may result in an underestimation of entropy. Both studies discussed above claim to detect underestimation in the Collision and Compression tests by comparing the values calculated by these two tests to the values calculated by the predictor-based tests [2][10]. Predictor-based tests use simultaneous sub-predictors to attempt to predict the next value in an output and then use the sub-predictor that is correct most often for the final calculation. The sub-predictor tests are lag, Lz78y, Multi MCW, and Multi MMC. Because NIST 800-90B primarily focuses on min-entropy, which measures unpredictability based on the probability of guessing the next output, it can be argued that the predictor-based tests hold more weight than the Collision and Compression tests, which do not perform calculations based on predictions. If results show that the Collision and Compression estimates are much lower than the prediction-based test results, it could be argued that overestimation occurred during CPU Jitter 3.6.1 testing.

# Methodology

This research calculates and cross-analyzes the min-entropy and NIST 800-90B NIID test measurements for four technical environments: Ubuntu WSL Container, Virtual Machine, Bare-Metal, and Bare-Metal Docker container. Testing is performed on CPU version 3.6.1 and utilizes the NIID test package found in NIST’s Entropy Assessment Github. Output provides min entropy levels and NIID measurements for 1 bit, 4, bit, and 8 bit testing.

## Required Materials

The following materials are required to complete the testing procedures in their entirety:

* Test laptop, virtual machine, host container/WSL
* Empty Flash Drive
* Rufus
* A CPU with high-resolution timer
* Docker account
* Ubuntu 24.04 LTS via bare-metal, container, and hypervisor environments
* Jitter Library Github at [GitHub - smuellerDD/jitterentropy-library: Jitterentropy Library](https://github.com/smuellerDD/jitterentropy-library)
* SP800-90B Entropy Assessment Github at [GitHub - usnistgov/SP800-90B\_EntropyAssessment: The SP800-90B\_EntropyAssessment C++package implements the min-entropy assessment methods included in Special Publication 800-90B.](https://github.com/usnistgov/SP800-90B_EntropyAssessment)

## Creating Ubuntu WSL Environment

1. Ensure containerization is enabled in the system’s BIOS settings.
2. Once enabled, in the command prompt, use command: wsl –install -d Ubuntu
3. This will initialize the installation and setup of Ubuntu on the host Windows machine

## Creating Virtual Machine Environment

1. Download a third-party application, such as Virtual Box.
2. Download the Ubuntu 24.04 LST image from the Ubuntu website.
3. Open the virtual box application and create a new machine.
4. Follow virtual box application instructions to select and download the Ubuntu image and set the parameters and user information for the environment.
5. Start the machine to begin the installation process.

## Creating Bare Metal Environment

1. With an empty flash drive connected, open a Rufus application and download the Ubuntu 24.04 LST image, as well as the drive to be formatted into a boot drive.
2. Click Start to begin the formatting process.
3. After formatting is complete, securely remove the drive and insert into the device the user wants to install a new operating system on.
4. On the device, boot into BIOD and then select the just-created boot drive as the device to boot from.
5. Boot the computer from this drive, initializing the installation process for the operating system chosen.
6. Follow instructions that appear on the screen, setting user information and other settings, to finalize the installation and creation of the bare-metal environment.

## Creating Bare-Metal with Docker Environment

1. In a bare-metal environment, remove all conflicting packages: *for pkg in docker.io docker-doc docker-compose docker-compose-v2 podman-docker containerd runc; do sudo apt-get remove $pkg; done*
2. Set up Docker’s apt repository: *sudo apt-get update sudo apt-get install ca-certificates curl sudo install -m 0755 -d /etc/apt/keyrings sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc sudo chmod a+r /etc/apt/keyrings/docker.asc # Add the repository to Apt sources: echo \ "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] https://download.docker.com/linux/ubuntu \ $(. /etc/os-release && echo "${UBUNTU\_CODENAME:-$VERSION\_CODENAME}") stable" | \ sudo tee /etc/apt/sources.list.d/docker.list > /dev/null sudo apt-get update*
3. Install Docker packages. Use the command: *sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin*
4. Verify successful installation by running the “hello world” image. Use the command: *sudo docker run hello-world.*
5. Start a new container for testing. Use the command: *sudo docker run -it ubuntu /bin/sh*

## Collecting CPU Data

Ensure the CPU has a high-resolution timer. This can be completed by using the command line. Enter the command: Check /proc/timer\_list (Linux). This file lists timer activity. User for tome activity using command: cat /proc/timer\_list | grep.

The following table describes the CPU data for each environment utilized in this study. To replicate this study, it is recommended to use the same CPU measurements and hardware architecture as defined below.

TABLE I. CPU Data

| Bare-metal CPU | Virtual Machine CPU | WSL Container CPU | Docker container CPU |
| --- | --- | --- | --- |
| Ubuntu 24.04 LTS Linux  Kernel version x | Virtual Box version x  Linux 6.11.0-21-generic | Windows Subsystem for Linux(WSL) version: 2.4.12 | Docker version x |
| Hardware CPU:  Architecture: x86-64  Intel Core i7-7700HQ, 2.80GHz, 4 Core(s), 8 Logical Processor(s) | Hardware CPU:  Architecture: x86-64 | Hardware CPU1:  Architecture: x86-64  AMD Ryzen 7 3700X 8-Core Processor, 3593 Mhz, 8 Core(s), 16 Logical Processor(s)  Hardware CPU2: | Hardware CPU:  Architecture: x86-64  Intel Core i7-7700HQ, 2.80GHz, 4 Core(s), 8 Logical Processor(s) |

## Building the Test Environment

For each environment, complete the following steps to download the required repositories from GitHub.

1. Open the Ubuntu Command Terminal
2. Create a Jitter directory. Use the command: *mkdir <directory>*
3. Verify that the directory was created. Use the command: *ls*
4. The directory created in step 2 should appear.
5. Navigate to the directory created in step 2. Use the command: *cd <directory>*
6. Download the Jitter Entropy Library from GitHub. Use the command: *git clone* [*https://github.com/smuellerDD/jitterentropy-library.git*](https://github.com/smuellerDD/jitterentropy-library.git)

NOTE: These instructions use version 3.6.1. To test other versions, download the appropriate repository.

1. Download the SP800-90B Entropy Assessment library from GitHub. Use the command: *git clone* [*https://github.com/usnistgov/SP800-90B\_EntropyAssessment.git*](https://github.com/usnistgov/SP800-90B_EntropyAssessment.git)
2. Download the Jitter RNG Daemon from GitHub. Use the command: *git clone* [*https://github.com/smuellerDD/jitterentropy-rngd.git*](https://github.com/smuellerDD/jitterentropy-rngd.git)
3. Use the command: *ls*
4. The directory should contain SP800-90B\_EntropyAssessment, jitterentropy-rngd-1.2.8, and jitterentropy-library-3.6.1
5. Use the command: *sudo apt-get update*
6. Download the required supporting packages. Use the command: *sudo apt install make*
7. Use the command: *sudo apt install gcc g++*
8. Navigate to the jitterentropy-library-3.6.1 directory. Use the command: *cd* *Jitterentropy-library-3.6.1*
9. List all files. Use the command: *ls jitterentropy-library-3.6.1*
10. A Makefile should appear.
11. Use the command: *make*
12. Use the command: *sudo make install*
13. To view debug information, use the command: *make -d*
14. After installation is complete, view multiple jitterentropy files. Use the command: *ls*

Optional - Not recommended within docker or podman container

1. Navigate to the jitterentropy-rngd-1.2.8 directory. Use the command: *cd  jitterentropy-rngd-1.2.8*
2. List all the files in the directory. There should be a Makefile. Use the command: *ls*
3. Make the Makefile. Use the command: *sudo make*
4. Use the command: *sudo make install*
5. Verify the jitterentropy.service file has been added. Use the command: *ls*
6. Add the file to systemd so it can be managed using systemctl. Use the command: *sudo cp jitterentropy.service /etc/systemd/system*
7. Use the command*: sudo systemctl enable jitterentropy.service*
8. Use the command: *sudo systemctl status jitterentropy*
9. Start the service. Use the command: *sudo systemctl start jitterentropy.service*

## Conducting Self Tests

1. Navigate to the SP800-90 B\_Entropy\_Assessment directory. Use the command: cd SP800-90B\_Entropy\_Assessment
2. Install the supporting packages. Use the command: *sudo apt-get install libbz2-dev libdivsufsort-dev libjsoncpp-dev libssl-dev libmpfr-dev*
3. Navigate to the cpp directory. Use the command: *cd cpp*
4. Use the command: *make*
5. Navigate to the selftest directory. Use the command: *cd selftest*
6. Run the self-test. Use the command: *./selftest*

## Collecting Test Data

1. Navigate to the Jitterentropy directory.
2. Navigate to the directory containing the invoke testing script.
3. Use the command: *cd tests/raw-entropy/recording\_userspace/*
4. Run the script: ./invoke\_testing.sh
5. Navigate back to the ../raw-entropy/validation-runtime folder
6. Run the command: sudo make
7. Edit the processdata.sh script using a text editor such as vi or nano to put the full path of your ea\_non\_iid location
8. In the /raw-entropy/validation-runtime folder, execute the command: vi ./processdata.sh
9. Edit the following lines to include the full path:
   1. EATOOL\_NONIID="../../SP800-90B\_EntropyAssessment/cpp/ea\_non\_iid"
   2. EATOOL\_IID="../../SP800-90B\_EntropyAssessment/cpp/ea\_non\_iid"
10. Save and exit the file: use *‘Esc’ :wq!*
11. Run the ./processdata script again. It should finish with the last printed output reading: Now analyzing entropy for conditioned data…

## Custom Binary Conversion Script

1. Download the provided analysis.zip folder and unzip it.
2. Once unzipped, move to the analysis folder by running the command: *cd analysis*
3. Once in the analysis directory, run the command cat readme.md. This will give brief instructions on how to run the provided script
4. Run the provided script by executing the command *./run\_analysis.sh [file path to jent-raw-noise-0001.data file]*
5. Once run, the script will convert the data into 1 bit, 4 bit, and 8 bit files, then ask for the file path to the ea\_non\_iid tool. This filepath is typically /SP800-90B\_EntropyAssessment/cpp/
6. The script will test each data file, and return the results of these tests to the output directory, along with the 1bit, 4bit, and 8bit binary files along with the raw data file.

# Results

The following sections summarize the results achieved during the cross-analysis: min-entropy, min-entropy with increased CPU load, and NIID tests. Discussion and analysis of results can be found in section V.

## Min-Entropy Results

The following table summarizes the min entropy measurements derived from each environment for 1 bit, 4 bit, and 8 bit tests. Results indicate bare-metal with container produced the highest level of min-entropy. Results are further analyzed in section V.A.

TABLE II. Min-Entropy

| Technical Environment | 1 bit | 4 bit | 8 bit |
| --- | --- | --- | --- |
| WSL Ubuntu Container | .319050 | 3.475193 | 6.447519 |
| Ubuntu Virtual Machine | .845001 | 3.515911 | 5.843111 |
| Ubuntu Bare-Metal | .838008 | 3.632568 | 6.748648 |
| Ubuntu Bare-Metal w/ Docker Container | .887651 | 3.548815 | 7.111979 |

## Min-Entropy with CPU Loading Results

The following table summarizes the min-entropy results with CPU loading. The number of background and system processes running in each environment can slightly impact the results. Testing for each environment is always recommended for security-based applications that may decide to embed this tool into their source. Results are analyzed in section V.B.

TABLE III. Min-entropy with cpu loading

| Scenario | Script | Stress-ng  Parameters | What the Load Is Doing | Min(H\_original, 8 × H\_bitstring) |
| --- | --- | --- | --- | --- |
| Baseline | None (only collect\_entropy.sh) | No additional system stress | System mostly idle – CPU clocks steady, caches stay warm, memory access patterns regular. | 5.395339   |  | | --- | |  | |
| CPU Load | cpu\_load.sh | --cpu 4 --cpu-load 90 --cpu-method all --timeout 60s | Keeps 4 CPU cores ~90 % busy for 60 seconds. | 5.518910 |
| Memory Load | mem\_load.sh | --vm 4 --vm-bytes 80% --vm-method all --vm-rw 50 --timeout 60s | Four workers allocate ≈ 80 % of RAM and repeatedly read/write it, 50-50, stress for 60 seconds. | 5.746277   |  | | --- | |  | |
| Mixed Load | mixed\_load.sh | --cpu 0 --cpu-load 90 --vm 2 --vm-bytes 50% --io 2 --hdd 1 --timeout 60s | Simultaneously stresses CPU 90% + RAM 50% + IO stress, 120 seconds stress. | 5.976806   |  | | --- | |  | |

## NIID Results

The following tables summarize the NIID test results for each environment for 8-bit, 4-bit, and 1-bit testing consecutively. Results are analyzed in section V.C.

TABLE IV. NIID 8-bit results

| Test | WSL Ubuntu Container | Ubuntu Virtual Machine | Bare-Metal Ubuntu with Docker Container | Bare-Metal |
| --- | --- | --- | --- | --- |
| Most Common Value | 7.456970 | 7.686075 | 7.517540 | 7.048585 |
| Collision Test Estimate | .8844907 / 1 bit | 1.00000 / 1 bit | .942282 / 1 bit | .929776 / 1 bit |
| Markov Test Estimate | .976134 / 1 bit | .997925 / 1 bit | .999726 / 1 bit | .999815 / 1 bit |
| Compression Test Estimate | .805940 / 1 bit | .730389 / 1 bit | .909821 /1 bit | .917849 / 1 bit |
| T-Tuple Test | 7.086406 | 7.086406 | 7.114833 | 6.748648 |
| LRS Test Estimate | 7.718814 | 7.522373 | 7.787702 | 7.539276 |
| MultiMCW | 7.664953 | 7.129230 | 7.575646 | 7.250280 |
| Lag Prediction | 7.764135 | 7.124044 | 7.43507 | 7.686376 |
| MultiMMC | 7.625916 | 7.362920 | 7.670420 | 7.142296 |
| LZ78Y | 7.625604 | 7.362898 | 7.670697 | 7.142068 |

TABLE V. nIID – 4 bit

| Test | WSL Ubuntu Container | Ubuntu Virtual Machine | Bare-Metal Ubuntu with Docker Container | Bare-Metal |
| --- | --- | --- | --- | --- |
| Most Common Value | 3.974198 | 3.977619 | 3.979858 | 3.971263 |
| Collision Test Estimate | .930623 / 1 bit | .920033 / 1 bit | .929895 / 1 bit | .925751 / 1 bit |
| Markov Test Estimate | .999584 / 1 bit | .997929 / 1 bit | .999507 / 1 bit | .999479 / 1 bit |
| Compression Test Estimate | .868798 / 1 bit | .878978 / 1 bit | .887204 / 1 bit | .908142 / 1 bit |
| T-Tuple Test | 3.677642 | 3.677642 | 3.876073 | 3.876073 |
| LRS Test Estimate | 3.520975 | 3.986078 | 3.872578 | 3.981737 |
| MultiMCW | 3.972900 | 3.995865 | 3.989220 | 3.988139 |
| Lag Prediction | 3.983108 | 3.986734 | 3.974356 | 3.779874 |
| MultiMMC | 3.989815 | 3.987813 | 3.998592 | 3.980176 |
| LZ78Y | 3.983879 | 3.987171 | 3.994245 | 3.991983 |

TABLE VI. NIID -1 bit

| Test | WSL Ubuntu Container | Ubuntu Virtual Machine | Bare-Metal Ubuntu with Docker Container | Bare-Metal |
| --- | --- | --- | --- | --- |
| Most Common Value | .992159 | .994181 | .995961 | .995518 |
| Collision Test Estimate | .879421 | .911818 | .898254 | .914443 |
| Markov Test Estimate | .991001 | .998721 | .998887 | .999505 |
| Compression Test Estimate | .777318 | .845001 | .887651 | .838008 |
| T-Tuple Test | .319050 | .921623 | .919089 | .939780 |
| LRS Test Estimate | .525946 | .995301 | .994600 | .995664 |
| MultiMCW | .395275 | .995991 | .993082 | .994892 |
| Lag Prediction | .388666 | .997459 | .996247 | .998505 |
| MultiMMC | .382267 | .997602 | .996954 | .994971 |
| LZ78Y | .388666 | .996768 | .998081 | .998848 |

# Discussion

The following sections provide analysis and discussion of the results presented in section IV. Overall, it is indicated that bare-metal with container environment produced the highest level of min-entropy, but the bare-metal alone produced more unpredictability. It was also indicated that CPU loading may impact min-entropy measurements.

## Min-Entropy Analysis

The closer the min-entropy value is to the bit value, the more optimal the min-entropy is considered. Optimal min-entropy ensures that data exhibits appropriate levels of randomness and unpredictability needed to meet cryptographic standards. As seen above, the Ubuntu Bare-Metal and Ubuntu Bare-Metal with Docker Container environments produced the highest levels of entropy of the four environments tested. Ubuntu Bare-Metal with Docker Container produced the highest min entropy levels for 8 bit and 1 bit testing, 7.119 and .8876, respectively. The Ubuntu Bare-Metal only environment produced highest min entropy levels for 4 bit, 3.625. The Ubuntu Virtual Machine produced appropriate min entropy for 1 bit testing, but produced lower entropy for the 8 bit testing, 5.843. Finally, the WSL container produced the lowest min-entropy levels overall at .3190, 3.475, and 6.4475.

Based on these results, it is recommended to utilize Ubuntu Bare-Metal with Docker for 1 bit and 8 bit, and to use Ubuntu Bare-Metal only for 4 bit. It is not recommended to use the WSL container for 1 bit and 4 bit, but may be used for 8 bit, but min-entropy levels will not be as optimal as in the bare-metal environments. The Ubuntu Virtual machine can be used for 1 bit and 4 bit, but is not recommended for 8 bit. These results are due to proximity with the underlying hardware and influenced by factors such as running background processing and memory access. In a bare-metal environment, the entropy source is much closer to the underlying hardware than in a containerized environment. Containerized environments also use fewer resources and may have fewer background processes running. This may greatly reduce the execution timings picked up by CPU Jitter to produce the entropy, resulting in lower min-entropy measurements.

## Min-Entropy with CPU Load Results

Based on the 8-bit min-entropy testing results from Ubuntu VM in Table , it is observed that system load significantly impacts the quality of entropy generated by the Jitter Entropy source. The baseline (idle system) condition produced the lowest min-entropy value of 5.395, indicating lower randomness. Introducing CPU load increased the entropy slightly to 5.519, suggesting that CPU stress alone adds minor timing variability. Memory stress further improved the entropy to 5.746, highlighting that high memory usage introduces greater micro-timing unpredictability. The highest min-entropy value of 5.977 was achieved under mixed stress conditions (simultaneous CPU, memory, and IO load), demonstrating that combining different system resource stresses resulted in maximizing entropy quality. Results data is presented in Table III.

## NIID Results

Similar to the min-entropy tests, Bare-Metal and Bare-Metal with Docker container environments produced the highest NIID measurements and the WSL container produced the lowest values. From this study, the WSL not only produced the lowest min-entropy, but likely also exhibits the lowest levels of randomness and unpredictability.

However, from a cross-analysis standpoint, the four environments varied substantially on which tests performed the highest or produced lower values. From the data, in most cases, each environment would show two consecutive tests displaying the highest produced value. For example, if the T-Tuple Test for an environment produced the highest value, then the LRS Test also produced the highest value, compared to other environments. This is likely due to both tests assessing averages in the data.

Overall, the WSL container produced highest results for MultiMCW and Lag prediction in 8 bit testing and produced highest results for Collision and Markov in 4 bit, but produced the lowest results overall for 1 bit. Similarly, the virtual machine produced highest values for most common value and collision estimate for 8 bit, MultiMCW and Lag for 4 bit, and Collision, MultiMCW and Multi MMC for 1 bit. Bare-metal with container produced the highest results for T-Tuple, LRS, and LZ78Y in 8 bit, Most common value, T-Tuple, MultiMMC and LZ78Y in 4 bit, and Most common value and compression in 1 bit. Finally, bare-metal produced the highest result in Markov, Compression, and MultiMMC in 8 bit, compression, t-tuple, LRS in 4 bit, and Markov, T-Tuple, LRS, Lag, and LZ78Y in bit 1.

Due the amount of variability in results, it is difficult to cross-analyze results by individual tests and overall highest values. However, some characteristics for each environment can be derived from the results. First, Bare-Metal with Docker container performed the lowest on predictor based tests versus non-predictors for each bit test. It can be concluded that while this has optimal entropy, the data may be more predictable compared to other environments. However, the bare-metal environment without a container, performed better on the predictor-based tests, but produced lower results for non-predictor tests. The virtual machine environment also produced lower results for predictive tests versus non-predictive, while WSL produced lower results for predictor and non-predictor test.

In combination with results on min-entropy, NIID results indicate that the bare-metal with container produced the highest min-entropy, but also produced lowest values for unpredictability. Similarly, bare-metal alone produced min entropy of 6.748646, but produced more unpredictability in the data. Based on this assessment, while bare-metal with container produced the highest values, bare-metal alone is the most optimal environment as it also produces high unpredictability, as shown in the NIID test results. Result data is presented in Table IV.

## Study Limitations

This research presents some limitations that can be further studied in later projects. First, due to time constraints, this research did complete full SP 800-90B compliance. To be considered fully compliant, CPU Jitter version 3.6.1 needed to undergo a series of restart and health tests. These tests detect any problems that may occur during the startup of an entropy source and if noise source failures and environmental changes are affecting the randomness quality of the data.

This research also did not conduct conditioning as specified in NIST 800-90B. After calculating the min-entropy of each environment, it was determined that applying conditioning would result in very similar output as the initial min-entropy estimate, indicating an insignificant result.

One limitation found during this research was the differing variables in the amount of background processes running on the test environments. Because CPU Jitter derives its entropy from CPU execution timings, the background processes running on a given environment may impact the amount of entropy produced from that environment. Similarly, only bare-metal and containerized environments were tested during this study. Future work may test additional environments such as AWS Cloud.

# Conclusion

Cross-analysis of CPU Jitter version 3.6.1 across container, virtual machine, bare-metal, and bare-metal with container environments found that bare-metal and bare-metal container environments produced optimal min-entropy as compared to the other environments. This is due to the closeness of the underlying architecture and increased processing, CPU, and memory access of a bare-metal environment that may not be present in other environments. NIID tests for unpredictability indicated that bare-metal and bare-metal with container environments also produced the highest values as compared to other environments.

However, this study is limited as it did not take NIST SP 800-90B restart tests, health tests, and conditioning into account. All of which must be tested for an entropy source to be considered compliant.

This research is significant because it establishes that CPU Jitter as an entropy source meets cryptographic standards only if it is used in specific environments, and that the environment an RNG or entropy source is utilized in may drastically impact its overall quality. This has not been previously studied in assessments of prior versions of CPU Jitter.

# Acknowledgements

This research was accomplished with consulting and guidance from Dr. Kerry McKay of National Institute of Science and Technology. Dr. Patrick Pape of the University of Alabama in Huntsville, College of Business, also guided students in completing this research.

# References

[1]smuellerDD. “SMUELLERDD/Jitterentropy-Library: Jitterentropy Library.” GitHub, [github.com/smuellerDD/jitterentropy-library](http://github.com/smuellerDD/jitterentropy-library) Accessed 31 Jan. 2025.

[2] Sönmez Turan, M., Barker, E., Kelsey, J., McKay, K., Baish, M., & Boyle, M. (2018, January 10). Recommendation for the entropy sources used for random bit generation. CSRC. <https://csrc.nist.gov/publications/detail/sp/800-90b/final>

[3] Müller, Stephen. CPU Time Jitter Based Non-Physical True Random Generator, 21 July 2022, [https://doi.org/https://www.chronox.de/jent/CPU-Jitter-NPTRNG.pdf](https://doi.org/https:/www.chronox.de/jent/CPU-Jitter-NPTRNG.pdf).

[4] “Random numbers from CPU execution time jitter” <https://lwn.net/Articles/642166/>

[5] Hagerty, Patrick, and Tom Draper. "Entropy bounds and statistical tests." Proceedings of the NIST Random Bit Generation Workshop, Gaithersburg, MD, USA. 2012.

[6] National Institute of Standards and Technology. SP800-90B\_EntropyAssessment. GitHub, 2023, <https://github.com/usnistgov/SP800-90B_EntropyAssessment>.

[7] Mahalingam, Hemalatha, et al. "Non-identical inverter rings as an entropy source: NIST-90B-verified TRNG architecture on FPGAs for IoT device integrity." Mathematics 11.4 (2023): 1049.

[8] Allagui, Anis, et al. "Parallel random bitstreams from a single source of entropy based on nonthermal electrochemical microplasma." Plasma Processes and Polymers 17.12 (2020): 2000123.

[9] Turan, Meltem. “CSRC Presentation: Overview of NIST RNG Standards (90A, 90B, 90C, 22).” CSRC NIST RNG Presentations, NIST, 31 May 2023, [csrc.nist.gov/Presentations/2023/overview-of-nist-rng-standards-90a-90b-90c-22](http://csrc.nist.gov/Presentations/2023/overview-of-nist-rng-standards-90a-90b-90c-22).

[10] Zhu, Shuangyi, et al. "Analysis and improvement of entropy estimators in NIST SP 800-90B for non-IID entropy sources." IACR Transactions on Symmetric Cryptology (2017): 151-168.