HyRNG: Hybrid Random Number Generator with Mouse Movement TRNG Seed and In-DRAM CSPRNG for Enhanced Hardware Security

Matthew Free, Hsiang-Pin Ko

Abstract—In this paper, we present HyRNG, a hybrid random number generator (RNG) that combines true random number generation (TRNG) from mouse movement with a cryptographically secure pseudo-random number generator (CSPRNG) integrated in-DRAM. The goal of this approach is to provide a fast, lightweight, and secure RNG suitable for hardware security applications, particularly in defending against memory-based attacks.

1 Introduction

Current In-DRAM RNGs (random number generators) face a tradeoff between slow, expensive generators of true-random numbers and fast, cheap generators of pseudo-random numbers. State-of-the-art mitigations for Rowhammer and Rowpress attacks rely on random numbers to determine which row to refresh. The RNGs that provide these numbers must be efficient enough to provide the required number of random values to the DRAM for use in mitigation in each refresh cycle. In-DRAM PRNGs (pseudo-random number generators) provide this efficiency; however, if the seed is determined, then the generator is deterministic and, therefore, can be exploited and defeated.

RNGs are either PRNGs, CSPRNGs (cryptographically-secure pseudo-random number generators, or TRNGs (true-random number generators). PRNGs generate values use a mathematical algorithm using an initial seed to generate numbers in a deterministic manner. CSPRNGs are PRNGs however they have cryptographic guarantees with their algorithm as it is resistant to reverse-engineering. TRNGs use no algorithm but instead use physical entropy to generate numbers, and therefore are completely unpredictable.

In this paper, we propose a novel scheme for an in-DRAM SPRNG that is seeded by a TRNG that uses hardware mouse movement. This results in a hybrid system that is fast, light, and generates non-deterministically random numbers.

2 Background

2.1 Mouse Movement TRNG

One robust source of a TRNG for PC users is mouse movement. Unlike physical entropy sources such as thermal or atmospheric noise, which require specialized hardware, mouse-based randomness can be harvested using standard input devices [1]. However, raw mouse movement data can exhibit patterns, especially when sourced from the same user over time. To address this, cryptographic post-processing is often applied to eliminate biases and improve entropy quality [1].

The most effective algorithm for this purpose is the MASK (Mouse movement-based Algorithm for Secure Key generation) algorithm [1]. MASK transforms collected mouse trajectories into binary image matrices and applies a nine-round image encryption process to scramble the data. This process results in a highly randomized output, which is then reduced to a 256-bit random number. MASK's design incorporates key cryptographic principles, such as confusion and diffusion, to enhance unpredictability and resistance to reverse engineering.

Among various mouse movement TRNG techniques, MASK stands out for its balance of security, speed, and entropy quality. It has been shown to outperform other algorithms in terms of efficiency and randomness, making it a strong candidate for user-centric entropy generation [1].

2.2 PRNG Attacks

While PRNGs offer efficiency and high throughput, they are inherently vulnerable due to their deterministic nature. If an attacker can predict or recover the internal state or seed, the entire output sequence becomes reproducible—compromising any system relying on randomness for security. This makes PRNGs a critical point of failure in cryptographic applications, secure communications, and system integrity protections.

Common attack vectors include seed guessing, poor entropy initialization, or side-channel attacks that leak internal state information through power, timing, or electromagnetic emissions. In hardware implementations, these risks are amplified when the seeding process is weak or when the PRNG lacks post-processing. Side-channel research such as *CrossTalk* [7] demonstrates practical exploitation of Intel's hardware-based PRNG instructions (RDRAND, RDSEED), using inter-core transient execution attacks to leak random values across cores–highlighting real-world vulnerabilities even in PRNG implementations on the market today [3].

These risks underscore the need for robust seeding using high-entropy sources (e.g., TRNGs) and cryptographic post-processing, as well as the importance of isolation and resistance to hardware-level leakage in a secure PRNG design.

2.3 Hardware CSPRNG Component

Hardware-based CSPRNG components provide efficient, high-throughput, low-latency random number generation with cryptographic security guarantees. These are often

integrated into memory controllers using secure algorithms such as AES-CTR, SHA-256, or ChaCha20 to generate high-entropy pseudorandom sequences [6]. Among these, ChaCha20 is particularly well-suited for hardware due to its stream cipher design, which avoids complex S-boxes and relies instead on lightweight ARX (Addition-Rotation-XOR) operations. This results in a low-latency, high-speed implementation with a smaller hardware footprint.

ChaCha20 operates on a 512-bit internal state and performs 20 rounds of transformations using a combination of key material, counter, and nonce. Its simplicity and parallelizability make it ideal for hardware CSPRNG use, particularly in hybrid TRNG-SPRNG designs where it can rapidly expand small amounts of entropy from a TRNG into secure, high-volume pseudorandom data streams.

Recent research, such as *Fortuna* [5], further supports the feasibility of embedding such secure CSPRNG mechanisms directly into memory controllers and DRAM systems.

2.4 In-DRAM TRNGs

In-DRAM True Random Number Generators (TRNGs) take advantage of inherent physical behavior in DRAM to derive entropy without adding special-purpose hardware. Conventional TRNGs typically rely on tailored circuits to capitalize on sources like thermal or atmospheric noise. By contrast, in-DRAM TRNGs generally use phenomena in commodity memory that occur in nature—like data retention errors, access latency fluctuations, and power supply noise—in order to directly derive randomness from commodity memory.

An exemplary illustration of such an approach is DRaNGe, which targets utilization of access latency variability as an entropy source [4]. In contrast to fault induction based on voltage tampering or refresh cessation, DRaNGe takes advantage of process variation and environmental noise to detect fine-grained differences in latency in DRAM cells. Such fine-grained timing variations, exacerbated by temperature fluctuation and manufacturing variation, lead to highly nondeterministic behavior that can be harvested to produce random bits. Its low-overhead integration with standard DRAM interfaces stands out, as does its ability to maintain high speed without compromising on memory performance or demanding special memory modules. It is free from complicated fault-induction techniques in its design, which makes it more pragmatic and energy-conserving in nature for security-related applications, particularly in systems with limited resources.

Although having advantages, in-DRAM TRNGs such as DRaNGe have challenges. Long-term entropy stability, aging effect sensitivities, and vulnerability to side-channel leakage are active research topics. With ongoing DRAM scaling, maintaining security guarantees through guaranteed randomness across device generations and manufacturing variation is paramount. Future implementations might have to decrease even more the energy-efficiencies of an in-DRAM TRNG to still provide security and performance in later device generations.

In addition to DRAM-internal physical phenomena, some in-DRAM TRNG implementations employ Linear Feedback Shift Registers (LFSRs) as lightweight digital post-processing tools to enhance randomness extraction. While LFSRs themselves are deterministic and typically unsuitable as standalone TRNGs, they can serve as effective entropy conditioners, whitening the noise harvested from DRAM behaviors. In this role, LFSRs help mitigate bias and temporal correlation in raw bitstreams, thereby improving statistical quality and entropy per bit. This allows the core entropy source—such as timing-based read errors—to remain unchanged while still meeting cryptographic randomness requirements. Their low hardware overhead and ease of implementation make LFSRs a natural complement to DRAM-based TRNGs in constrained environments.

3 Implementation

As outlined in our project proposal, we aimed to complete the following tasks by the end of our project:

1) Set up the simulation environment
2) Evaluate TRNG seed generation approaches
3) Implement the most suitable TRNG method
4) Create a ChaCha20 component
5) Implement in-DRAM generation of encrypted random numbers using the TRNG-CSPRNG hybrid approach

We selected Ramulator, an open-source DRAM simulator, as our simulation environment—motivated in part by its prior use in in-DRAM TRNG work such as DRaNGe [4].

3.1 TRNG Evaluation and Implementation

We explored multiple TRNG seed generation approaches, including thermal noise and RowHammer-induced bit flips. However, Ramulator lacks built-in support for thermal noise simulation, and implementing a custom thermal noise model is beyond the scope of this project. Similarly, RowHammer-based techniques were deemed inappropriate, as our goal is to use randomness to help mitigate RowHammer vulnerabilities—not to exploit them.

As a result, we identified mouse movement as the most practical and effective TRNG source. We adopted the MASK algorithm proposed by Yue Hu et al. [1], which converts user mouse movement into binary images and encrypts them into 256-bit random seeds.

We have implemented this method in a Python script that actively collects real mouse movement data and generates TRNG seeds. These seeds are stored and streamed into Ramulator to serve as both initial inputs and periodic reseeds for the CSPRNG component.

3.2 CSPRNG Integration

We have developed a ChaCha20-based CSPRNG as a C++ module within Ramulator. This implementation supports key initialization using a 256-bit (32-byte) key and a 96-bit (12-byte) nonce, consistent with standard ChaCha20 specifications. Once initialized, the cipher can generate an arbitrary amount of pseudorandom data. This is not per-bank, as we are mainly focusing on the efficiency of this system, and can concern ourselves about the per-bank nature of the design later on.

In the Ramulator main function, we read TRNG-generated seeds from the mouse movement stream, use them to initialize the ChaCha20 cipher, and produce encrypted random numbers. This completes our hybrid TRNG-CSPRNG pipeline, enabling in-DRAM pseudorandom number generation based on real-time entropy.

3.3 Periodic Reseeding

The final aspect of our solution is the periodic reseeding. As CSPRNG algorithms are not as random as TRNG solutions, we wanted to improve the way we generate numbers with CSPRNG. Instead of an initial set seed, which could be prone to having low entropy, we implement periodic reseeding of the initial seed we feed into the ChaCha20

module. At the end of the set period, the TRNG seed will be replaced with a new seed. In theory, this value will be ready whenever the algorithm requires a reseed. It is important to note that whenever if the TRNG module does not produce a new seed in time for the interval, the old seed will be used until the end of the next period.

4 RESULTS

We configured the simulation using the following parameters and assumptions:

- **TRNG Access Latency**: Accessing a TRNG value should take 1 cycle. As we are assuming the value will be ready at any moment, even if the reseeding interval is not sufficiently long. In this model, the value is simply loaded from a register or buffer, similar to Intel's *RDRAND* instruction, which takes one cycle if the random value is ready [2].
- **CSPRNG Generation Latency**: To generate a CSPRNG output using the ChaCha20 algorithm is assumed to take 23 cycles. This estimate is obtained from a open-source hardware implementation available on GitHub [9].
- **TRNG Reseeding Interval**: The reseeding interval is calculated based on the TRNG design presented by Hu et al. [1]. We are assuming that a TRNG generation time of 1 ms on an 800 MHz processor, this equates to 800,000 cycles. The paper reports a generation time of 78 ms, which would equate to 62,500,000 cycles; however, this value includes the time required for image generation, which our implementation does not require. We instead use a simplified matrix-based calculation without image generation to reduce the reseeding interval.

4.1 NIST Tests

For testing randomness of the various portions of our system, we utilized the NIST test suite, specifically, STS (SP800-22) for the CSPRNG values (with and without reseeding) and SP800-90B for the TRNG values. For our TRNG values, this was collected using the MASK algorithm in the background. We were able to record 40000 bits of data, which amounted to 1200+ TRNG values. This is shown in Table 1, where it can be seen that the TRNG currently does not meet the bar for a reliable TRNG ($h_{\text{Bitstring}} \geq 0.997$). This means that the TRNG, with the limited data we were able to provide, as opposed to the 1000000 that are needed for a reliable result, can provide 0.8392 bits of entropy in an IID (independently and identically distributed) context and 0.5983 bits of entropy in a non-IID context.

Metric	IID Result	Non-IID Result
Assessed Entropy (h_{Assessed})	8.000	4.7867
Min-Entropy per Bit ($h_{\text{Bitstring}}$)	0.8392	0.5983
Original Entropy (h_{Original})	6.2341	—
Passed Chi-Square Test	No	No
Passed IID Permutation Test	No	—
Passed Longest Repeated Substring	Yes	—

TABLE 1: SP800-90B Entropy Assessment Results for TRNG

The CSPRNG values were obtained through running the simulation with the TRNG values as seeds. For both, the workload we ran them on resulted in 175 Mb of data which we streamed as 175 1 Mb bitstreams across the 15 tests of the full NIST test suite for randomness. The results are shown in Table 2 for without reseeding and Table 3 for with reseeding.

NIST Test Name	P-value	Proportion	Status
Frequency (Monobit)	0.0000	0/175	Fail
Block Frequency	0.0000	133/175	Fail
Cumulative Sums	0.0000	0/175	Fail
Runs	0.0000	0/175	Fail
Longest Run of Ones	0.0000	173/175	Fail
Rank	0.1574	172/175	Pass
Discrete Fourier Transform	0.0169	175/175	Pass
Non-Overlapping Template	0.067	129/175	Fail
Overlapping Template	0.0000	125/175	Fail
Universal Statistical	0.0000	14/175	Fail
Approximate Entropy	0.0000	0/175	Fail
Serial (1st Order)	0.0000	0/175	Fail
Serial (2nd Order)	0.3950	174/175	Pass

TABLE 2: NIST STS test results for the HyRNG bitstream.

NIST Test Name	P-value	Proportion	Status
Frequency (Monobit)	0.000000	0/175	Fail
Block Frequency	0.000000	133/175	Fail
Cumulative Sums	0.000000	0/175	Fail
Runs	0.000000	0/175	Fail
Longest Run of Ones	0.000001	173/175	Fail
Rank	0.157355	172/175	Pass
Discrete Fourier Transform	0.016854	175/175	Pass
Non-Overlapping Template	0.002554	133/175	Fail
Overlapping Template	0.000000	125/175	Fail
Universal Statistical	0.000000	14/175	Fail
Approximate Entropy	0.000000	0/175	Fail
Serial (1st Order)	0.000000	0/175	Fail
Serial (2nd Order)	0.395016	174/175	Pass

TABLE 3: NIST STS results for HyRNG with reseeding.

For each test, P-values are collected, where the null hypothesis for each test is that a perfect RNG would not have produced random data with better characteristics for the given test than the tested streams [8]. If the resulting P-value for the test is greater than our chosen level of significance $\alpha = 0.01$, we accept the null hypothesis for the test. For the test that passed, we note that ≥ 169 of the 175 bitstreams passed the test with similar P-values.

From these tables, we can see the HyRNG system does not perform as expected. The acceptable range of passing sequences if $[0.98, 1]$, of which both cases are significantly far from. This suggests that the system produces somewhat predictable, partially random values.

4.2 Latency with Ramulator

Using Ramulator, we tested our HRNG solution on performance across four benchmark traces. Ramulator reports the latency of memory read operation in terms of cycles. For each benchmark trace, we ran a simulation under three different configurations: (1) the baseline Ramulator, (2) with HRNG enabled, and (3) with HRNG and periodic reseeding.

Benchmark	No HRNG	HRNG	HRNG with reseeding
gcc	231.931	257.709	257.710
leela	178.733	197.800	197.801
linpack	239.319	262.916	262.917
mcf	193.227	210.781	210.782

TABLE 4: Average memory latency cycles per request for all read requests.

Across the four benchmark traces, we see that the latency increased about 10% with HRNG enabled. However, introducing reseeding increased the latency by a negligible amount, increasing the the latency by less than 0.01 cycles in every case.

4.3 Raw Random Bit Generation Throughput

To evaluate the performance of the HRNG design, we calculate the rate of which HRNG can generate random numbers. We then compare this with previous solutions.

$$\text{RNG Throughput} = \frac{\text{Total Random Bytes Generated}}{\text{Total Time}}$$

With the ChaCha20 algorithm, HRNG can output 128 bits of random data each iteration. Note that the reseeding interval is set at 800,000 cycles to ensure that a TRNG value always available.

Benchmark	Throughput
TRNG (mouse movement) only	256 b/s
HRNG without reseeding	4452.17 Mb/s
HRNG with reseeding	4452.17 Mb/s
D-RaNGe [4]	717.40 Mb/s

TABLE 5: Comparison of the rate of which random numbers are generated.

We can see that although TRNG generate rate is really low, it is expected. This is why we introduced HRNG by combining the slow TRNG with CSPRNG using ChaCha20 algorithm. We see that our HRNG solution is **six times** as fast at generating random numbers compared to past solutions like D-RaNGe.

5 Discussion

While our HyRNG solution had better throughput and fairly low increase of latency, our NIST evaluation revealed its shortcoming of the quality of its random output. Yet, due to the integration of a CSPRNG component to generate randmo numbers, we were able to achieve speeds better than prior in-DRAM RNG solutions such as D-RaNGe by over sixfold. Additionally, the latency overhead introduced by HyRNG was relatively modest ($\sim$10%), and periodic reseeding latency was negligible.

However, the NIST test suite results really showed the limitation of our solution. Neither the TRNG nor CSPRNG aspect met the statistical thresholds expected for industry standards. The TRNG component, based on the MASK algorithm, fell short of the required entropy levels ($h_{\text{Bitstring}} \geq 0.997$), achieving only 0.8392 (IID) and 0.5983 (non-IID). These entropy values were likely propagated into the CSPRNG, undermining its effectiveness.

Furthermore, across multiple NIST STS tests, the hybrid RNG failed a significant number of trials. These failures were consistent across tests such as Frequency (Monobit), Runs, and Approximate Entropy. Although a few tests like Rank and Discrete Fourier Transform passed, the overall result suggests that there was a lack of quality randomness and potential predictability. Notably, reseeding did not improve these results either, which could indicate that either the seed quality or the CSPRNG integration method is insufficient for ensuring long-term entropy integrity.

Several factors may have contributed to these outcomes:

- Limited sample size in the TRNG collection phase (only 40,000 bits) is likely the key issue of the entropy evaluation.
- The MASK algorithm, while cryptographically sound in theory, may require more user diversity or longer collection times to get a better dataset.
- Our simplified model disregard potential entropy dilution due to noise, timing, or buffering constraints in a realistic DRAM controller environment.
- The implementation of our ChaCha20 algorithm is might not be fully accurate. Such a standard encryption algorithm should easily pass the NIST test, yet from our findings, it failed to do so. This suggest that we might have not implemented the algorithm correct, and instead, should have imported from a known library instead.

6 Conclusion

In this work, we proposed HyRNG, a hybrid RNG system that combines user-driven entropy (via mouse movement) with an efficient in-DRAM ChaCha20-based CSPRNG to deliver high-throughput, low-latency random number generation. While the architecture met our performance and integration goals, it failed to satisfy statistical randomness criteria outlined by NIST standards.

Our main concern lies in the entropy quality of the TRNG source and its propagation into the overall system. Although our design significantly outperformed prior work in speed and introduced minimal performance overhead, it currently lacks the reliability needed for cryptographic or high-assurance applications.

Future work should explore enhanced entropy conditioning, greater TRNG data collection, alternative or multiple entropy sources, and more robust reseeding mechanisms. With these improvements, we believe the HyRNG architecture could evolve into a practical and secure solution for memory-integrated RNG in hardware security contexts.

7 Source Code

Here is the full source code of our project which was originally a fork of Ramulator: https://github.com/mko02/CS7292_HRNG.

References

[1] Y. Hu, X. Liao, K. wo Wong, and Q. Zhou, "A true random number generator based on mouse movement and chaotic cryptography," *Chaos, Solitons Fractals*, vol. 40, no. 5, pp. 2286–2293, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0960077907008958

[2] Intel Corporation, "Intel® 64 and IA-32 Architectures Software Developer Manuals," 2024, accessed: 2025-04-30. [Online]. Available: https://www.intel.com/content/www/us/en/developer/overview.html

[3] B. Kaliski and P. Kocher, "The intel random number generator," Intel Corporation, Tech. Rep., 1999. [Online]. Available: https://www.rambus.com/wp-content/uploads/2015/08/intel-rng.pdf

[4] J. S. Kim, M. Patel, H. Hassan, L. Orosa, and O. Mutlu, "D-range: Using commodity dram devices to generate true random numbers with low latency and high throughput," pp. 582–595, 2019.

[5] R. McEvoy, J. Curran, P. Cotter, and C. Murphy, "Fortuna: Cryptographically secure pseudo-random number generation in software and hardware," in *2006 IET Irish Signals and Systems Conference*, 2006, pp. 457–462.

[6] J. Pfau, M. Reuter, T. Harbaum, K. Hofmann, and J. Becker, "A hardware perspective on the chacha ciphers: Scalable chacha8/12/20 implementations ranging from 476 slices to bitrates of 175 gbit/s," in *2019 32nd IEEE International System-on-Chip Conference (SOCC)*, 2019, pp. 294–299.

[7] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "Crosstalk: Speculative data leaks across cores are real," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1852–1867.

[8] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert *et al.*, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," National Institute of Standards and Technology, Tech. Rep. Special Publication 800-22 Revision 1a, 2001, https://csrc.nist.gov/publications/detail/sp/800-22/rev-1a/final.

[9] Secworks, "ChaCha - A hardware implementation of the ChaCha stream cipher," https://github.com/secworks/chacha, 2024, accessed: 2025-04-30.