7-2-2024

DESIGN AND IMPLEMENTATION OF TRULY RANDOM NUMBER GENERATION USING MEMRISTORS FOR IN-MEMORY COMPUTING

Nick Felker
Rowan University

DESIGN AND IMPLEMENTATION OF TRULY RANDOM NUMBER GENERATION USING MEMRISTORS FOR IN-MEMORY COMPUTING

by

Nick Felker

A Thesis

Submitted to the
Department of Electrical & Computer Engineering
College of Engineering
In partial fulfillment of the requirement
For the degree of
Master of Science in Electrical & Computer Engineering
at
Rowan University
December 12, 2023

Thesis Chair: Dwaiypan Chakraborty, Ph.D., Professor, Department of Electrical & Computer Engineering

Committee Members:
Dr. John Schmalzel, Ph.D., Professor, Department of Electrical & Computer Engineering
Dr. Sangho Shin, Ph.D., Principal Engineer, TBT Group

Dedication

Dedicated to my family, who spent many hours shuttling me between New York City and Glassboro to attend classes.

Acknowledgement

I want to acknowledge Dr. Shin, my original thesis advisor and the professor who introduced me to the rich field of memristors.

Abstract

Nick Felker
DESIGN AND IMPLEMENTATION OF TRULY RANDOM NUMBER GENERATION USING MEMRISTORS FOR IN-MEMORY COMPUTING
2023-2024
Dr. Chakraborty, Ph.D.
Master of Science in Electrical & Computer Engineering

This paper proposes a new security module based on non-volatile memory. The module uses a memristor-based true random number generator to generate random numbers which can be used for cryptography. The module is implemented in software using a modified RISC-V instruction set architecture.

The paper evaluates the performance of the module using the RISC-V simulator Gem5. The results show that the module can generate random numbers at a rate of 63 microseconds per number, which is faster than the standard C library's random number generator. The module can also be used to scramble strings of characters and generate hashes of inputs.

Table of Contents

Table of Contents (Continued)

Table of Contents (Continued)

List of Figures

Figure Page

List of Tables

Table Page

Chapter 1

Introduction

1.1 Random Number Generators

Generating and using keys is an important method of creating secure connections between trusted systems on a network. Cyberattacks often try to masquerade with malicious system requests, so networks can be secured by encrypting requests using securely generated keys.

Private and public key systems rely on the generation of random numbers of a predetermined bit-lengths. However, current random number generators are not truly random, rendering them susceptible to exploitation by malicious actors who can predict the random number pattern or by stealing these keys. Exploiting these vulnerabilities can allow attackers to authorize requests disguised as a trusted system.

There are many different approaches to random number generators. One method is to use a Gaussian Random Number Generator (GRNG) which selects numbers at random based on a Gaussian distribution [1]. Alternatively one may prefer to use a Uniform Random Number Generator (URNG) which is designed to pick random numbers across a given range with equal probability.

The Wallace method can be used to efficiently generate random numbers from a normal distribution by starting with a uniform distribution and transforming that into its position in an inverse cumulative distribution. It is fast but can be less accurate compared to a Box-Muller transformation [2].

Previous research has focused on novel techniques to improve the randomness and unpredictability of the generator. RAVA is one proposed design which uses two reverse-biased Zener diodes to generate sufficient amounts of noise as the input to a generation

algorithm [3]. The noise is then measured in a 3 microsecond window and quantized to high or low. While this approach is easy to implement, the throughput is relatively slow and is not resilient against environmental and circuit attacks by malicious actors.

In this paper we propose a new security module based on non-volatile memory. We use memristors, which are non-volatile devices, to achieve true randomness through non-determinstic means [4]. Our module generates truly random numbers faster than contemporary systems, making it a promising solution for secure key generation in low-powered embedded systems.

We document the software requirements to achieve truly random number generation and how it could integrate into modern software applications. We contend this will perform a significant improvement in security across the industry.

1.2 Memristors

The memristor is a passive hardware element first proposed by Chua [5] in 1971 and realized by HP in 2008 [6]. These devices have a dynamic resistance which can be controlled by passing in voltages. When a high voltage is passed across, its resistance changes, allowing it to be modified. By programmatically controlling the reads and writes, the device's resistance can be quantized and interpreted as a binary value.

$$d\phi = M \cdot dq \tag{1}$$

There are many proposed kinds of materials which behave like a traditional memristor in its "pinched hysteresis loop" [7]. The memristance is defined as a relationship between magnetic flux and capacitance above (Equation 1).

$$M(q) = R_{off} \cdot \left(1 - \frac{\mu \cdot v \cdot R_{on}}{D^2} \cdot q(t)\right) \tag{2}$$

With high voltage, the impedance of a memristor will fluctuate between a high

resistance R_{off} and a low resistance R_{on} [8]. This relationship can be defined as a function of charge as shown above (Equation 2).

1.2.1 Kinds of Memristors

Memristors represent a new class of non-volatile memory devices which exhibit resistive switching behavior. This means that their resistance can be changed by applying an electric field or current, and the new resistance state will be retained even after the field or current is removed. Within this class, there are multiple kinds of physical devices that have this behavior.

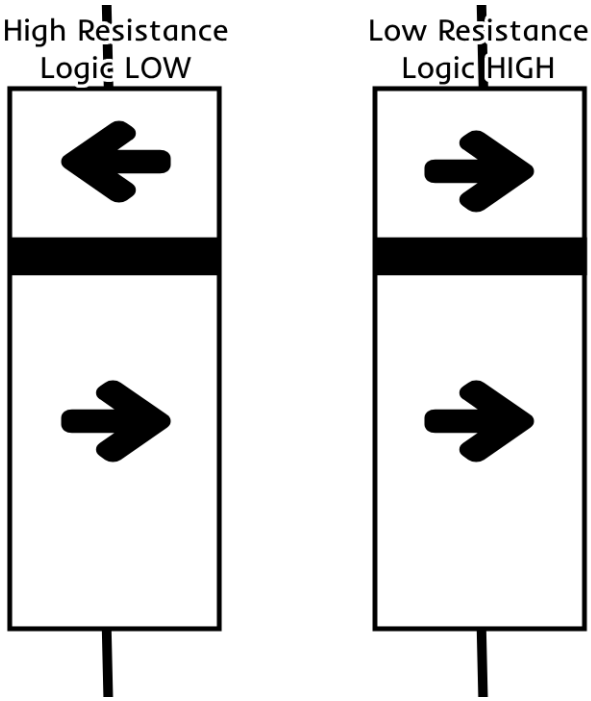
1.2.1.1 Magnetoresistive Random-Access Memory Magnetoresistive Random-Access Memory (MRAM) is a type of non-volatile memory which exploits properties of magnetism to store data [9]. Two ferromagnetic plates are used for each memory cell. One plate is fixed at a specific magnetic polarity while the other plate's magnetism can be changed in order to represent a binary value.

1.2.1.2 Phase-Change Memory Phase-Change Memory (PCM) is another kind of non-volatile memory. When specific electric currents are passed into a memory cell, a heating element can cause chalcogenide glass to become amorphous or enter one of many crystal states [10]. Its multiple kinds of states could allow a single cell to hold more than one bit of data [11].

1.2.1.3 Spin Transfer Torque Magnetic Tunnel Junction One particular hardware device is referred to as either STT-MTJ (Spin Transfer Torque Magnetic Tunnel Junction) or STT-MRAM (Spin Transfer Torque Memristive Random Access Memory) [12]. Each one has two magnetic layer with particular orientations, as shown in Figure 1. Passing a current through the thin magnetic layer, it can permanently flip the orientation. Thus, the orientation of each one can be read as a binary value.

Figure 1

STT-MRAM Has Two Layers Which Can Have Opposing or Aligned Magnetic Orientations



1.2.2 Evaluating and Selection of Memristor Hardware

Each type of memristor has its own advantages and disadvantages. MRAM is the most mature technology and is already being used in some commercial products. It is fast, energy-efficient, and has a long endurance. However, it can be expensive to produce. PCM is less mature than MRAM, but it has the potential to be much cheaper and denser. However, it is slower and less energy-efficient than MRAM.

STT-MTJ is the newest type of memristor and is still under development. It has the potential to combine the best features of MRAM and PCM, but it is still very expensive and difficult to produce [13]. This research proceeds with the assumption of improved manufacturing capabilities.

1.3 Realization of Memristors

Memristors are a relatively new class of devices, and there is still much research to be done on how to best realize them in physical systems. One approach is to develop mathematical and circuit models that can be used to simulate memristor behavior.

Kahale et al. propose a mathematical model for memristors, creating a MATLAB program to simulate them [8]. Their model is based on accurately measuring the electrical properties, particularly at the edges of the hysteresis loop.

Many circuit models of memristors have been proposed. Kvatinsky proposed a general memristor model called ThrEshold Adaptive Memristor Model (TEAM) [14] and later Voltage-ThrEshold Adaptive Memristor Model (VTEAM) [15]. This research focuses on creating models implemented as SPICE models, a common framework for circuit simulation. Using three memristors, one is able to form a 2-input NOR gate.

Another approach to realizing memristors is to develop hardware emulators that can be used to simulate the behavior of memristors in real time. Luo et al. developed a hardware emulator of non-volatile memory specifically to create a spiking neural network [16]. Their emulator is based on a Verilog model of memristors, which can perform matrix math with high performance.

Kim et al. built upon that research this with a FPGA-based model which is able to perform a greater variety of applications including fast fourier transformations and AES encryption [17].

1.3.1 Physical Realization of Memristors

The goal is to have physical memristor devices which are available in large quantities for manufacturing in devices such as smartphones and IoT devices. There is a lot of ongoing research into production for commercial applications.

One promising approach is to use crossbar arrays of memristors to create logic gates

and memory devices. Crossbar arrays are simple and scalable, and they have the potential to be used to create high-performance, low-power computing systems [18]. Chakraborty et al. examined sneak-paths in crossbar architectures, where signals travel through incorrect paths and can lead to data errors [19]. This requires second-order design in order to minimize the risk of mistakes.

1.4 Applications of Memristors

As models have developed, they have been used in pursuit of specific applications. With fast, low-power, small memory systems, a lot of research has targeted specific kinds of applications.

1.4.1 Neural Networks

Memristors are particularly well-suited for use in neuromorphic computing systems, which are inspired by the human brain [20]. Neuromorphic computing systems are able to learn and adapt in a way that is similar to how the human brain works. This makes them ideal for applications such as machine learning and pattern recognition. Schuman et al. examine novel algorithms which can run in neuromorphic computers [21].

The pinched hysteresis effect of memristors has been studied in relation to building neural networks. In addition to Luo et al, other research has been done on this subject.

Joshi et al. studied the use of memristors in generalized resistive RAM (ReRAM) accelerators for deep learning [22]. They examine the benefits but also analyze non-idealities. One example they study is PUMAc, which uses a custom instruction set architecture (ISA) with a custom compiler in order to optimize the use of registers and optimize for power. Their research also looks at TraNNsformer, optimized for training AI models and GENIEx, which is a simulator for non-ideality conditions.

Lazaro et al. propose a physical simulator built on top of a 1.2V LTSpice model called KNOWM, implemented in a real memristor development kit in order to study clas-

sifier models [23].

In order to study the usage of memristive memory in neural networks and deep learning, an open-source software package called MemTorch has been developed [24]. It is written on top of the PyTorch framework for Python-based machine learning applications [25]. This framework uses simulations of memristors, creating a virtual crossbar representation in order to research novel applications and use-cases.

In addition to developing simulations and models, more recent work has focused on manufacturing real hardware. Yao et al. presented a convolutional neural network (CNN) chip implemented with 2048 real memristors to develop a five-layer CNN in order to achieve high accuracy on MNIST image recognition [26].

An experiment performed last year by Wan et al. developed NeuRRAM, a physical RRAM-based chip which uses compute-in-memory to perform machine learning tasks accurately with great improvements in energy efficiency [27].

1.4.2 Edge Computing and Other Applications

Edge computing is a paradigm in which data is processed and analyzed at the edge of the network, rather than being sent to a central cloud server. Memory that requires a small amount of area and no power consumption in a passive state makes it work well for Internet of Things (IoT) applications. Specifically, the use of memristors has been studied in a number of edge computing applications.

Huanhuan et al. studied the use of memristors to accelerate the ShuffleNetV2 image classification algorithm on low-power devices [28]. They found that memristors can be used to achieve significant performance improvements over traditional hardware accelerators.

Donato et al. share research on the input/output behaviors of non-volatile memory to reduce errors, energy consumption, and speed [29]. They propose a system called MEMTI which uses a multitask learning technique to improve the performance of an image

classification model.

As IoT devices have low power requirements, Pala et al. propose a system called Freezer which caches device state in non-volatile memory as a power cycle occurs [30] in order to achieve two orders of magnitude improvement in backup time.

Research into the machine learning capabilities of memristors could enable AI in edge computing applications without requiring a lot of power. Contemporary AI on the edge sensors require large batteries in the hundreds of milliamp-hours, which then necessitate large and expensive charging circuits [31]. Reducing costs and power could enable a broader adoption of sensors in a greater number of applications.

Tan et al. recently processed a convolutional neural network (CNN) using a 16-level ReRAM in-memory compute architecture to serve as the core for an electronic nose, which takes in gas and concentration data to identify the kinds of odors in the air [32].

For data-rich applications such as mixed reality (XR), edge computers need to be able to process a lot of data in a small, wearable form factor while consuming a small amount of power [33].

The behavior of memristors present other kinds of applications which have been less studied up to this point. First, they could serve as a new form of analog memory and analog devices such as amplifiers, filters, and digital-to-analog converters [34]. Shin et al. demonstrated the use of memristors to build a midband differential gain amplifier [35]. Memristors have also been shown to be used in bandpass filters [36].

This could lead to the development of new electronics which are more efficient and smaller than contemporary hardware. Also, they can be connected together to form logic gates and other kinds of circuit components.

Research has also been conducted into its resilience to space-based radiation, which can cause single-event upsets [37, 38]. The ”MemSat” project was a CubeSat which launched with the aim to test the reliability of memristor-based memory compared with standard and radiation-hardened NAND flash memory.

1.4.3 Software Implementation of Memristors

Software implementations for memristors and non-volatile memory have been slower to develop than hardware implementations. This is because memristors are a relatively new technology, and there is still much research to be done on how to best use them in software systems.

Having fast memory which is also non-volatile can impact a lot of applications and software libraries. As memristors are non-volatile, they retain their state even after power is removed. This can be beneficial for some applications, but requires larger rewrites of these applications. Memaripour et al propose a framework called Breeze for non-volatile memory management which is able to achieve some minor performance improvements [39].

Another challenge of implementing memristors in software is that they are analog devices. This means that their state can vary continuously, rather than being discrete like the state of a traditional memory cell. This requires fine-tuning of analog-to-digital hardware and the need to develop new error correction algorithms [40].

Ye et al. propose a transparent approach where the first two bits of a memory address can be encoded with the desired memory architecture without requiring large rewrites [41].

Agrawal et al. studied the usage of memristors in stochastic computing [42]. Stochastic computing is a type of computing that takes advantage of the probabilistic nature of analog devices. Agrawal et al. showed that memristors can be used to perform stochastic operations more efficiently than traditional hardware.

Abreu presents new challenges to programming stochastic computer systems, as they must be resilient to inherent noise and their distributed nature [43].

Sliper et al. proposed a software journaling framework for tasks in intermittent computing systems [44]. Intermittent computing systems are systems that are frequently powered off and on. Sliper et al.'s framework allows tasks to be resumed intelligently

even if the system is powered off before the task is complete. This generally is known as persistent computing [45].

In addition to designing circuits to efficiently perform in-memory computing, software interfaces will need to be designed to receive memory and instructions from a central processing unit (CPU). Not enough research has focused on instructions that fit into modern instruction set architectures (ISA) which are consistent but also can be extended to work with this new domain of analog signals [46]. Additionally, improvements to program compilers will need to updated to incorporate these instructions.

1.4.4 File Systems

Memristors have the potential to revolutionize the way that file systems are designed. Traditional file systems are designed for volatile memory, which means that all data is lost when the power is turned off. Memristors, on the other hand, are non-volatile, which means that data can be retained even when the power is off.

Wu et al. explored the use of memristors in file systems for intermittent computing systems [47]. Intermittent computing systems are systems that are frequently powered off and on. Wu et al.'s research showed that memristors can be used to create file systems that are able to resume operations automatically after a power cycle.

When a low-power, potentially batteryless device, is rebooted, it must be able to resume those tasks automatically.

Most memristor applications are studied with two states: high resistance and low resistance. However, the device resistance can change along a spectrum between both ends. This could be used to create denser storage by using a memristor cell to store multiple bits. An analog-to-digital converter connected to a memristor which supports additional middle tiers of resistance would enable a multi-bit cell.

One experiment by Rao et al. created an 11-bit conductance level manufactured onto a real CMOS chip. This enables a single memristor to store 2048 bits of data reliably

[48]. Improved performance per memristor cell can enable this hardware to perform well at a greater scale even if manufacturing each one is more difficult compared to standard NAND flash today. This would also have performance improvements for area size and power.

1.4.5 NVM-Only Systems

Research on memristors and their applications in computing have touched upon the idea of an NVM-only system [49]. Given the benefits of non-volatility, what would an operating system look like where there was only non-volatile memory instead of standard RAM?

NVM-only systems have numerous theoretical advantages over a traditional memory system including power efficiency, resilience, and performance. Storage-class memory is the name given to this class of low-power non-volatile memory architectures [50].

One challenge is that NVM-only systems require new operating systems and software that are designed to take advantage of the unique properties of NVM. Another challenge is that NVM is still a relatively new technology, and it can be more expensive than traditional memory. Despite many promising factors, it has yet to prove itself in real systems.

NVM memory systems haven't achieved speeds comparable to contemporary RAM [51]. Still there is plenty of research on improving computer architectures to take advantage of this technology.

1.5 Processing In-Memory

STT-MRAM circuits are an active area of research for being useful for memory while also having low power and area requirements. Unlike DRAM, it is non-volatile and thus doesn't require constant power refreshing. These benefits allow it to serve as a viable candidate for next-generation memory.

As computing evolves, with new chips promising smaller sizes and faster speeds, the current computer architecture design has revealed bottlenecks that have slowed down improvements. The "Von Neumann bottleneck" [52] describes the issue where moving data from memory to processing requires a lot of energy and slows down the pace of execution. Memristors offer a way to circumvent this bottleneck.

Memristors can be laid out in a crossbar structure and used to create logic gates [53]. A memory device that can also perform logic would enable certain computer operations to be executed entirely within the memory unit. This in-memory processing [52] would then serve as an alternative for some instructions rather than the Von Neumann architecture.

The kinds of operations that can be performed in-memory is an area of active research. Song et al. recently proposed ReFloat, which uses resistive RAM to perform floating point operations and matrix-vector multiplication in the analog domain [54]. By removing the computation bottleneck and centralizing memory and compute operations, they were able to achieve an order of magnitude improvement in performance.

1.6 Secure Enclaves

Secure enclaves are dedicated hardware environments or coprocessors where trusted code can be executed [55]. This may be integrated into existing CPUs such as Intel's Software Guard Extensions (SGX) or an external hardware peripheral where users can securely authorize operations using their credentials [56]. These environments are critical for ensuring executable operations are verified.

New organizations like the FIDO alliance building new standards for secure requests over the Internet. These standards use second-factor hardware authentication and a transparent, passwordless system to improve trust and reduce the risk [57].

In all of these cases, there is a need to use modern cryptography and random numbers to generate keys which are unpredictable by malicious actors. As software systems are unable to be truly random, they will use various deterministic values which seed the

number generation algorithm which can be exploited [58]. Slow processors mean it is possible for an attacker to exploit the system by controlling the moment a random number is generated.

A seed is supposed to be a value which sets the initial state in pseudo-random number generators [59]. The kinds of values used should be difficult for attackers to guess. If they have the initial seed, they will be able to generate the same numbers as you. Often a system timestamp is used, although it requires a sufficient amount of resolution to match the desired entropy.

In one recent cryptocurrency application, a bug made it easy for attackers to break the random number generator to steal private keys [60]. This wallet generated a passkey with only 32 bits, a small amount of entropy. Additionally, the input seed was incorrectly entered, making it easy to exploit.

Another recent security vulnerability uses a lattice. This is essentially a structure which contains a series of vectors in vector space, which can often be likened to a specifically-constructed matrix or a graph. It can be useful in crytographic applications. Here, it was used to break RSA private keys due to bugs in the hardware implementation [61].

Other RNG bugs have made it trivial to determine both future and previous numbers, which can reveal a large number of internal secrets to a potential attacker [62]. There is a trade-off between communication performance and security. If the total number of bits are too low, attackers are able to brute force a small number examples to obtain private keys [63]. However, greater entropy comes at a cost of a slower speed in generating each cryptographic key.

The security concerns are amplified in IoT systems, where devices require minimizing power and area. This results in slower processors which are often unable to perform state-of-the-art cryptography. New paradigms like backscattering can minimize the need for on-device power [64], but then requires processing to be low-power as well. Addition-

ally, small cheap sensors do not have much space for additional expensive components.

Major attacks on networks of devices have been exacerbated by poor security practices including re-use of passwords and out-of-date cryptography [65]. There's a strong need to create a cryptographically secure random number generator [66].

Memristors and STT-MRAM are promising technologies to address contemporary challenges of security and performance in embedded systems. Using them to generate truly random numbers through secure processors should allow IoT devices to be made more secure and efficient.

1.7 Encryption

Cryptography as a field requires a way to securely generate keys for both encryption and decryption. This is critical to transmit private content over the Internet. As our data needs continue to grow, it is important that the amount of computing time dedicated towards encryption does not similarly grow.

Memristors have the potential to be used to improve the performance and security of encryption algorithms. Memristors are non-volatile memory devices that can be used to create physical random number generators (PRNGs). PRNGs are essential for generating secure keys, and memristor-based PRNGs have the potential to be more secure and efficient than traditional PRNGs.

Arumi et al. proposed a current-starved ring oscillator with an odd number of non-volatile memory inverters, which would alternate an output signal of HIGH and LOW, which then becomes an input for the next output. With a non-deterministic high resistance value, the oscillator frequency becomes unpredictable and thus chaotic [67]. By adding a transistor, the system is able to control the drain current and the oscillation.

1.7.1 Image Encryption

Image encryption is one extension of cryptography which involves converting an image into noise predictably so that it can be decrypted back to the original source only by having the original keys.

Previous research has investigated topics including using neural networks to encrypt and decrypt images with a high degree of sensitivity while also performing well in terms of megabits-per-second. Bigdeli et al. propose using a Hopfield neural network which uses three secret parameters and chaos functions to do this task reliably [68].

Lin et al. build on this research by using a theoretical memristor network to accomplish the same tasks [69]. Das et al. build a memristor-based image encryption system which supports colored images [70]. Their encryption algorithm uses truly random numbers to rotate and scramble small segments of the image. This workflow is shown below in Figure 2. The algorithm is shown below in **??**.

Chen et al. examine a faster image encryption algorithm which uses pseudo-random number generation and a 480-bit external key [71]. However, relying on an external pseudo-random component can present additional security risks that can be mitigated through an integrated design.

Figure 2

The Operations to Perform Image Encryption With Memristors

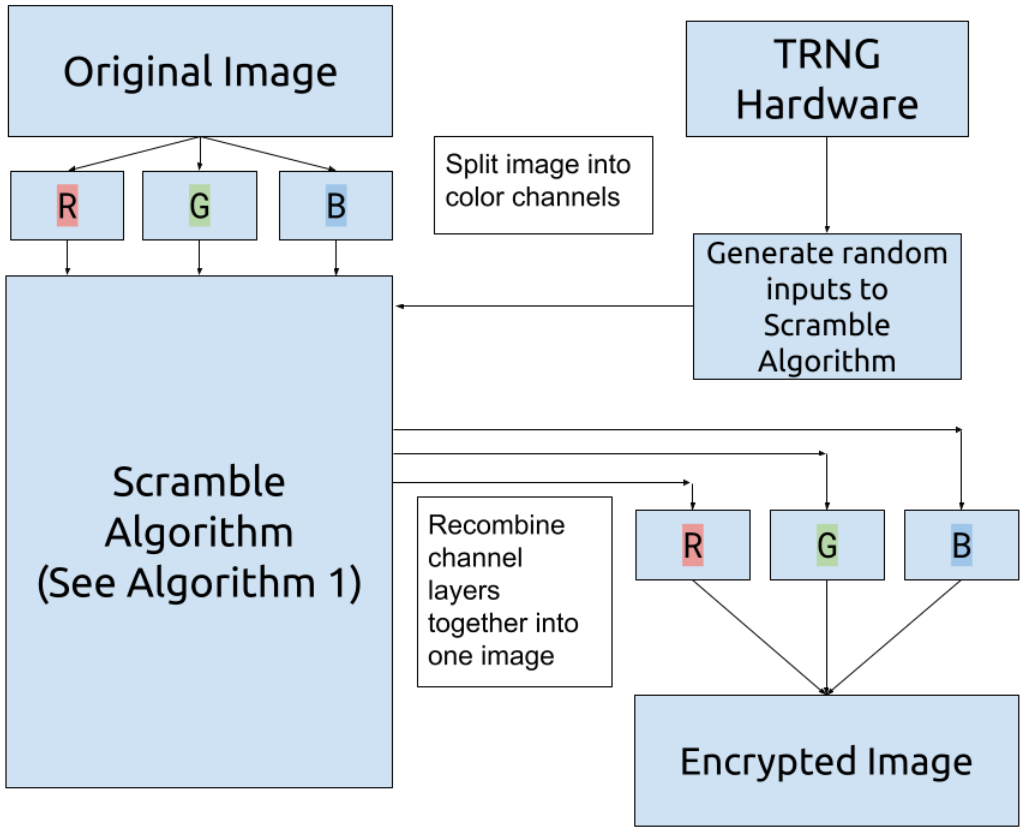


Figure 3

Algorithm 1. For Scrambling and Encrypting an Image

Image Encryption Scramble Algorithm

Create 32x32 pixel segments
Rotate each segment
Split each segment into 8x8 pixel blocks
Rotate each block
Scramble the position of pixels in this block
Repeat for every segment

Chapter 2

Design & Implementation

2.1 Hardware Circuit

The proposed hardware implementation for the security module is based on the STT-ANGIE design by Perach et al. [4]. This circuit consists of a capacitor, N memristors, and N sense amplifiers. Specifically they choose STT-MRAM as the hardware for the memristors due to their low power requirements. This is shown in Figure 4.

There are three stages to perform this number generation. First, the processor charges the capacitor to V_{init}. Next, it discharges the capacitor across the N memristor cells. Finally, the sense amplifiers read the output of each memristor. A sense amplifier generally is an electronic circuit which is able to magnify the difference in voltage of two nodes [72]. This amplified voltage can then be quantized to a binary value. Each output, being a binary value, can be concatenated together to form an N-bit number. This workflow is shown in Figure 5.

The individual material properties of each memristor cell cause the output voltage of each memristor to be slightly different in a way that is non-trivial to replicate perfectly. It also is difficult for physical analyzes of individual cells by observing the physical state. This resistance to common side-channel attacks results in a truly random number which cannot be easily obtained by an attacker.

Other designs for memristive random number generators explore specific kinds of distributions. Dong et al. present a memristor-based Gaussian random number generator, though only at the hardware level [1].

Figure 4

Hardware Diagram of a Truly Random Number Generator Using STT-MRAM

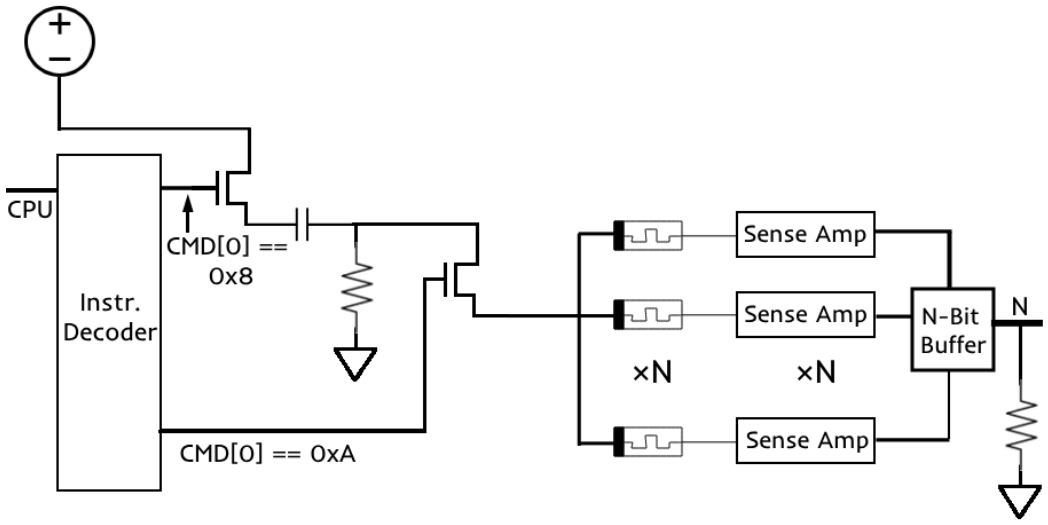


Figure 5

Operation Flowchart for Generating an N-Bit Random Number

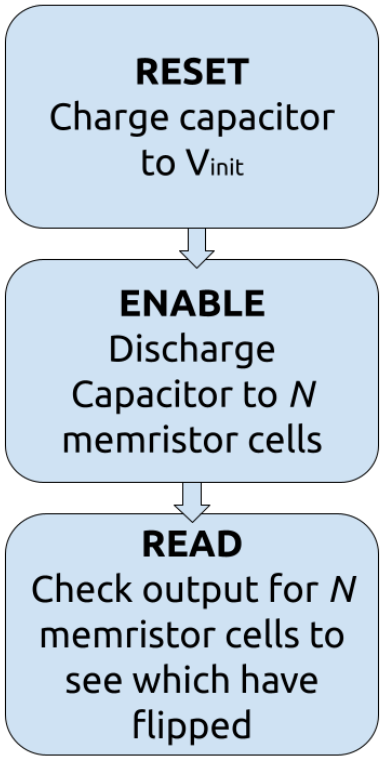


Table 1

Added RISC-V Instructions

Instruction	Machine Code	Description
TRNG.RESET	0x8006027	Charges capactior
TRNG.ENABLE	0xA006027	Discharges caps to memristors
TRNG.READ	0xC006027	Reads impedance of memristors

2.2 Instruction Set Architecture

To implement the hardware circuit in software, three new instructions were added to the RISC-V instruction set architecture [73]: *trng.reset*, *trng.enable*, and *trng.read*. Each instruction is meant to take a single clock cycle. These instructions and opcodes are shown in Table 1.

The *trng.reset* instruction is implemented as J-type instruction which takes a single numerical parameter. This input value is used as the seed. This is useful for simulation purposes but can also serve as an optional modifier to the voltage which will charge the capacitor.

The *trng.enable* can also be implemented as a J-type or U-type instruction although it has no parameters. This instruction discharges the capacitor into each memristor in the row.

The *trng.read* can be implemented as any instruction with a destination register. This can also be a J-type instruction with no parameters. This instruction reads the value of each memristor, quantizes it into a binary value, and then concatenates them into a single N-bit number which is placed into R_d.

A single user-level C-lang function can be written which executes these three assembly instructions in sequence by taking a pointer and setting the value to the output of the *trng.read* instruction.

2.3 Cryptography Functionality

The proposed security module can be used to implement a variety of higher-level cryptographic functions. For example it can generate hashes, scramble strings of characters, and generate one-time passcodes.

Based on the MemHash paper, a function was developed to perform a one-way scramble function on a string of characters in userspace [74]. A 6-bit character encoding was defined using primarily alphanumeric characters and assorted punctuation. For each character in the provided string, a random number is generated. Then the modulus is taken in order to generate the output character.

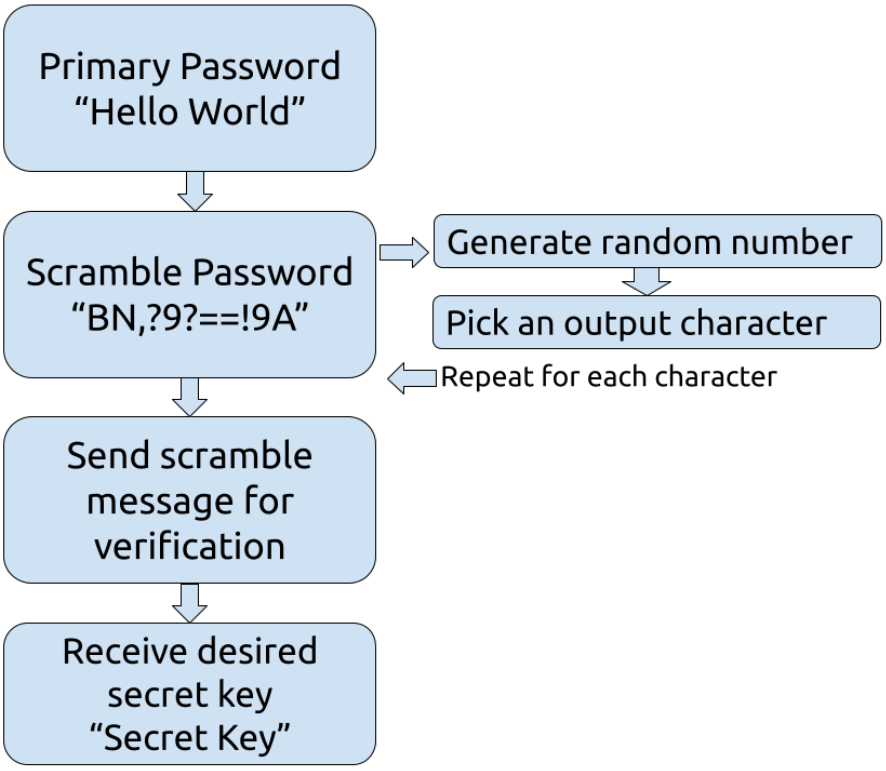
Generating hashes of inputs can serve as a key to unlock privileged data. Another function is written which takes in a hashed input. If that input matches the expected encrypted string, it will output the desired user secret key. Otherwise, it will output an error message.

For example, consider a password manager application which stores a collection of secret passwords in a secure memory space. To access any one of these passwords, it requires a primary password. However, for security purposes this password should be encrypted in transit. The password can be encrypted before being checked.

As shown in Figure 6, if my primary password is $PASSWORD$, this $scramble$ function may first generate the hashed password $BN,?9? ==!9A$. The secrets manager will check this against its own stored hash with the same value. Only then will it respond with the expected output.

Figure 6

Flowchart Where a Primary Key Can be Securely Exchanged for Secret Key



Chapter 3

Results

3.1 Simulations Using Gem5

To test the performance of our proposed security module, we used the RISC-V simulator in Gem5 [75]. The Gem5 simulator creates a reproducible, deterministic environment which can run binaries compiled with the modified RISC-V toolchain. A small change to the Gem5 environment allows for implementations of our three custom instructions [76]. Software models for the capacitor, memristor cells, and memristor controller were created. The controller was defined to generate 16-bit numbers, with each cell having slightly different memristance values to represent these material non-idealities. The code for these changes in Gem5 is shown in Appendix A.

One quirk is that the Gem5 environment is very deterministic, such that the $rand()$ function will return the same results on each test run. To add some variety, the $trng.reset$ introduces a seed parameter which is provided to the $srand()$ function to create more variety.

This test configuration simulated a 1 GHz RISC-V processor with DDR3 memory which, when started, will only execute the test program and then shut down. Table 2 shows the specific parameters chosen. In Gem5 these configurations are represented by a Python file which includes a path to a binary executable to run.

A test program was defined which could perform the operations described above. Two versions were developed. One emulates the truly random number generator using these custom instructions. The other uses the standard C library to perform the same actions. Optimizations were disabled when compiling each program for fair results.

Table 2

Gem5 Test Parameters

Property	Value
Memory Ranges	512MB
CPU	RiscvTimingSimpleCPU
DRAM Type	DDR3-1600
DRAM Capacity	512MB
DRAM Config	8x8
Clock Speed	1 GHz

3.2 Random Number Generation

In this program, 16-bit random numbers were generated and printed for N iterations. The total time of program execution within Gem5 for both programs was measured. As the Gem5 simulation was set to run at 1GHz, the times are defined in nanoseconds. As shown in Table 3, the MTJ program quickly becomes faster as more random numbers have to be generated.

Running these tests more granularly, it shows that each random number generation in the standard case takes *69 microseconds* while taking *63 microseconds* in the MTJ case. This means that more random numbers could be used while maintaining existing performance, which can help with security.

3.3 Password Retrieval

In a second program, a user-generated password is hashed for N iterations. The total time of program execution with Gem5 for both programs was measured. The code for this program is shown in Appendix C.

Table 3

Performance of Generating N Random Numbers

Iterations	CPU (ns)	STT-MTJ (ns)	% Difference
5	427923	411351	96.1%
6	497013	472206	95.0%
50	3551399	3273333	92.2%
100	7013797	6454726	92.0%
150	10474907	9626550	91.9%
200	13946042	12802170	91.8%

As shown in Table 4, the standard case continues to outperform the MTJ case for any number of iterations. Scrambling an 11-character phrase "*Hello World*" takes *2.0 milliseconds* in the standard case while it takes *1.93 milliseconds* in the MTJ case for each iteration. This allows more phrases or longer phrases to be scrambled at contemporary performance.

The difference in speed does reach a bottleneck quickly as a lot of CPU time is taken by performing the array lookups for the scramble operations and relatively little time is spent generating random numbers.

3.3.1 Ideal Bit Length

The ideal number of bits to generate is worthwhile investigating as well. The number of bits in the Gem5 simulator was adjusted from 1-32, doubling in each iteration. The amount of time it took to run 5 iterations versus 6 iterations was checked and then the difference was used to obtain the per-iteration value.

Figure 7 and Figure 8 below shows the length of time to generate a single N-bit random number or scramble a single character using an N-bit system. As shown from the

Table 4

Performance of Scrambling N Messages

Iterations	CPU (ns)	STT-MTJ (ns)	% Difference
5	10052966	9741418	96.9%
6	12043380	11669562	96.9%
50	97973371	95672377	97.7%
100	195846534	191248596	97.7%
150	293719696	286824815	97.7%
200	391592859	382401034	97.7%

simulations, the more bits used in the MTJ architecture leads to slightly longer performance times but still manages to be faster than current architectures up to 16 bits. The number of bits did not meaningfully change the performance of standard systems.

In a production system, the number of bits would be used to generate each character of an encrypted message. Keys with a greater amount of entropy would generate a random value for every bit of the final key.

3.4 Measuring and Comparing Randomness

To measure the randomness of algorithms, the National Institute for Statistics and Technology (NIST) have developed different statistical tests that aim to measure entropy. The frequency test can be run on a large dataset of numbers [77].

A short program was written to generate 1024 random numbers in both configurations. See Appendix B for the code. These numbers were then converted to binary. A count of the ones and zeros of each binary number was taken using a short JavaScript program. See Appendix D for the code. Both values were set to generate 32-bit numbers. The distri-

Figure 7

Performance of Random Number Generation by Bit Length

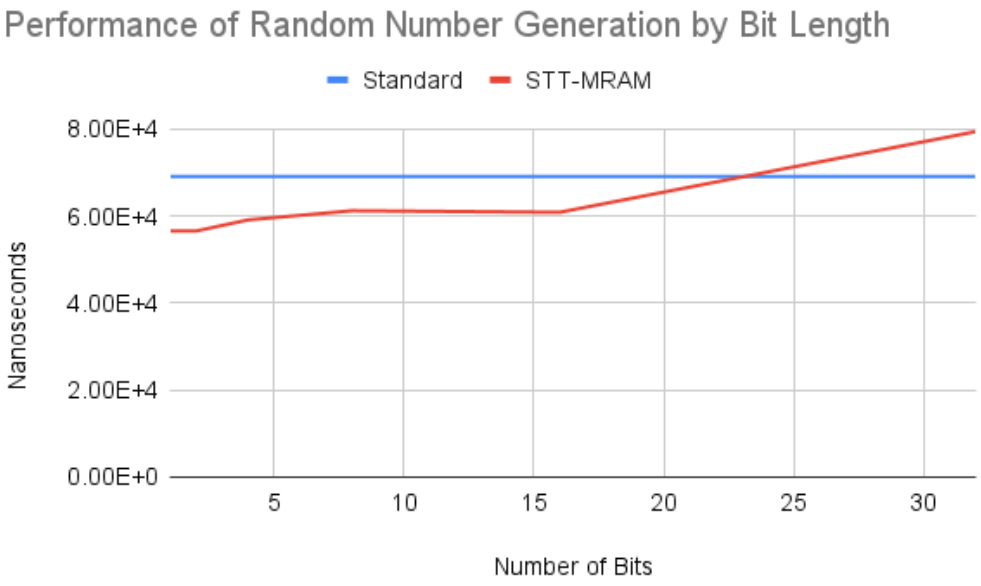
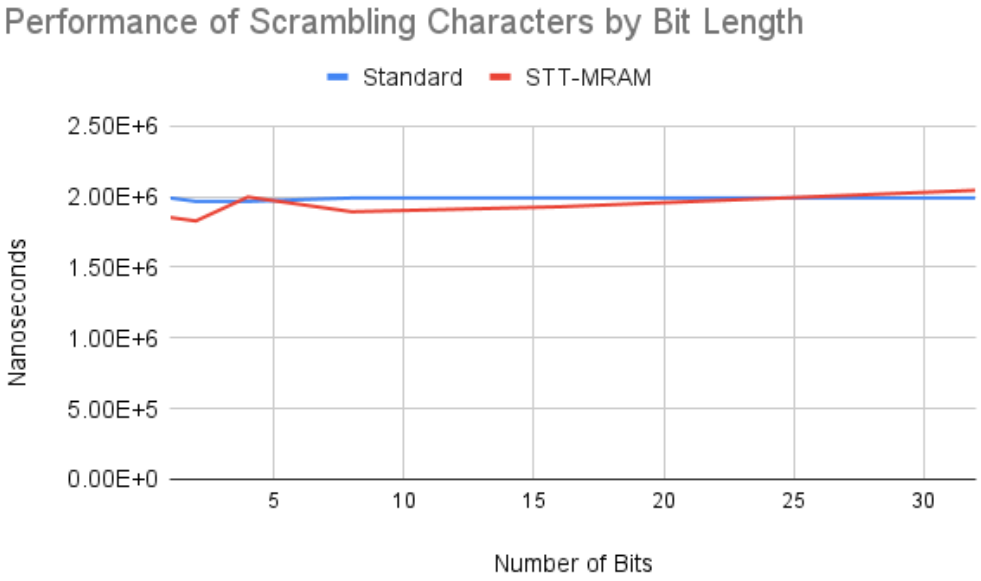


Figure 8

Performance of Scrambling Characters Generation by Bit Length



bution of numbers using the standard Gem5 CPU configuration is shown in Figure 9. The distribution of numbers using the STT-MRAM configuration is shown in Figure 10.

In a perfectly uniform, random distribution, the ratio of ones to zeros should be exactly 0.5. We can see from Table 5 that both the CPU and STT-MRAM cases have averages close to 0.5.

We next must calculate a ratio between the summation S_n and the square root of the number of entries n, where $n = 1024$ in our test case. S_n is defined as the sum of the total count of ones subtracted by the total count of zeros as shown in Equation 3. In a perfectly uniform, random distribution, these counts should cancel out. In a random sample, this value will not be perfectly matched and the value of S_{obs} is meant to statistically verify that this number is within expected bounds.

We then calculate a statistical value S_{obs} which is shown below in Equation 4.

The statistical test is shown in Equation 5. The value of S_{obs} must be lower than ϕ^{-1}, which here represents the percent point function of the standard normal distribution. The value of α is set as 0.01 as our p-value.

Given the test conditions, S_{obs} must be less than 0.840 to be considered sufficiently random. The results are shown below in Table 6.

We can see that the STT-MTJ is statistically shown to be random, 23 percent lower than the value calculated for the standard CPU case. However, we also see that the value for that case is higher than our expected threshold.

As noted above, Gem5 generates the same random numbers each time the simulation begins as it is very deterministic. This probably means it is not a very good platform for testing randomness in a system. It may be true that Gem5's implementation of random numbers is not sufficiently random. At the same time, we do see that our STT-MTJ system is much better even in this case.

$$S_n = \sum O_i - \sum Z_i \tag{3}$$

Figure 9

1024 Random Numbers Generated by Gem5

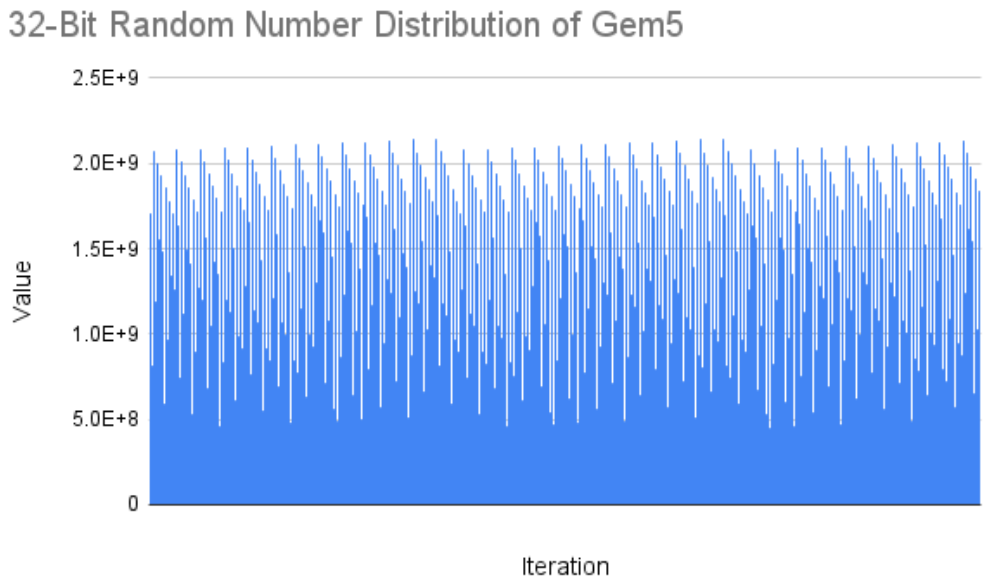


Table 5

Statistical Measurements of Random Number Sets

Config	Average	Standard Deviation
CPU	0.484	0.089
STT-MTJ	0.487	0.078

$$S_{obs} = \frac{|S_n|}{\sqrt{n}} \tag{4}$$

$$S_{obs} < \phi^{-1}(1 - \frac{\alpha}{2}) \tag{5}$$

Figure 10

1024 Random Numbers Generated by STT-MRAM

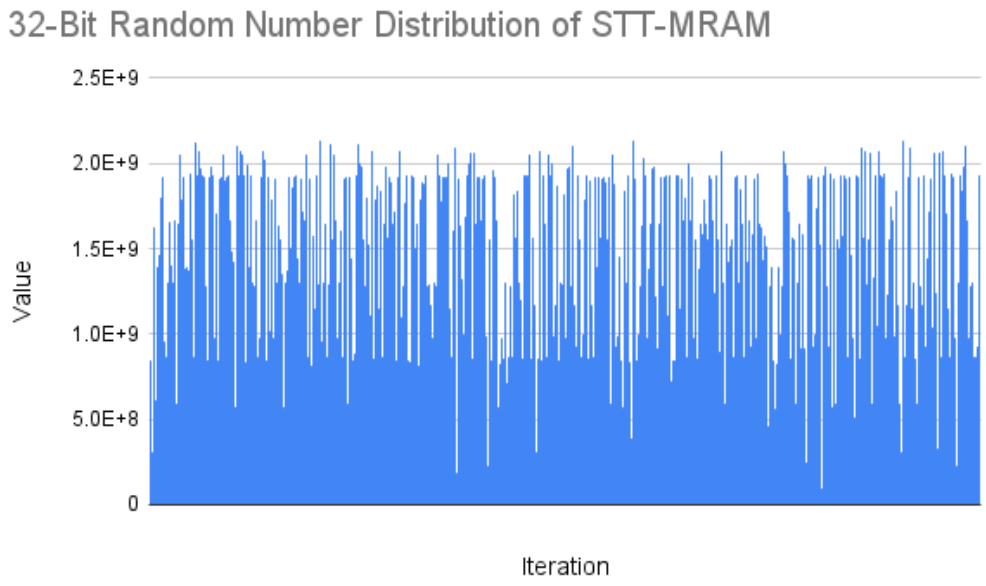


Table 6

Statistical Calculations of Random Number Sets

Config	S_n	S_{obs}
CPU	32	1.009
STT-MTJ	25	0.783

Table 7

Statistical Measurements of Random Number Sets

Config	Average	Standard Deviation
CPU	0.484	0.089
STT-MTJ	0.487	0.078
Windows	0.501	0.088

Table 8

Statistical Calculations of Random Number Sets

Config	S_n	S_{obs}
CPU	32	1.009
STT-MTJ	25	0.783
Windows	6	0.186

3.4.1 Expanded Randomness Testing

After these initial results, we can expand the systems we use to measure and compare entropy. The C program run in Gem5 was recompiled to run on a Windows 10 PC, outside of the sandboxes and virtual environments. This program was then run to generate random numbers using the CPU. The average and standard deviation are shown in Table 7.

The Windows execution environment is much closer to a perfect average, suggesting that Gem5 was providing a barrier even in the STT-MTJ test. These random numbers can then be processed in the same way as above to obtain values of S_n and S_{obs}, as shown in Table 8.

We see that the Windows CPU is much more random than the Gem5 CPU. It may require a physical realization of this system to verify whether the 23 percent improvement

in entropy is still applicable.

Chapter 4

Discussion

Getting consistent improvements to operation time can allow for more complex random number generation in the same amount of time. For additional security, the number of bits can increase to provide more entropy and reduce the likelihood of brute force and birthday attacks. Additionally, hashing operations can support longer phrases without reducing performance.

It should be acknowledged that the number of bits in a standard processor do not affect time where they do when using a memristive system. This may be an artifact of transferring register data back to the program and could be improved through future compiler optimizations. At the same time, it appears that 16 bits is an appropriate size to achieve performance benefits while maximizing entropy.

4.1 Physical Characteristics

In addition to performance benefits when using this in applications, there are benefits in terms of power and physical area. The physical size of an N-bit TRNG using STT-MRAM is roughly $400\mu m^2$ while using as little as $6pJ$ per-bit [4]. This demonstrates good performance with minimal hardware impact.

These results also show good results for low-end IoT devices. Additional security with minimal area and power requirements can allow them to provide improved encryption when uploading and downloading data in order to minimize the potential risk of an attack.

Further research may focus on ways to improve the circuitry in order to improve the performance of random number generation. Song et al. investigate ways to optimize the behavior of the sense amplifier in order to maximize the speed of generation [78].

4.2 Security Implications

Being able to generate truly random numbers makes it unlikely that attackers are able to use RNG manipulation to guess the values of keys. As the randomness depends on process-based differences to increase entropy, an attacker cannot use easily-obtainable methods to break the RNG.

More research will be needed to examine and mitigate potential attacks which may manipulate the environmental factors of memristors and change their characteristics in ways that may weaken their non-determinism. For example, changes in the ambient temperature can alter the ON/OFF state resistances of memristor cells [1]. Taken to more extreme temperatures, it could cause a way to generate deterministic values. Another factor to study is how the manipulation of supply voltage could cause a deterministic output [67].

Chapter 5

Future Work

I envision a truly-random number generator having the most benefit for Internet of Things sensors, which will require cryptography to send encrypted data to a base station without consuming a lot of power or using a lot of physical area.

A fleet of low-cost sensors can measure soil moisture on large farms, then send their results to an irrigation system to provide the perfect amount of water for crops. Precision agriculture can improve yields and reduce food waste without needing more land [79].

Cities can deploy sensors to monitor traffic, both pedestrian and vehicles. Dense environments can often become congested, and measuring usage can help urban planners understand what work needs to be done to improve resident quality of life.

These sensor networks can send data over Low-Powered Wide Area Networks (LoWPAN), which enable signals to be sent and relayed over many miles [80]. In cities, they can send data over 5G networks.

Additionally, low-powered sensors can take advantage of ambient broadcasts and use backscattering to remove the need for batteries. Data can be safely cached in memristors as memory cells through intermittent computing, then broadcast when it has accumulated enough power.

5.1 Cryptographic Integration

Future work should investigate standardized cryptographic and hashing algorithms to incorporate them into this memristor processing unit through in-memory computing. In particular, encrypting image and video frames efficiently is an active area of research. There are a number of companies creating security cameras and market cloud-based analyses of video content. AI vision is also coming to more gadgets including robot vacuums [81]. The

sensitive nature of these videos mean high-quality encryption is necessary for consumers to trust their data is secure.

It might be possible that existing algorithms are not ideal for memristor processing units. In that case, security experts should work with NIST to develop new standards along with tests to measure randomness.

5.2 Firmware Improvements

The results are demonstrated for 1-bit memristor cells. Future work could investigate performance at higher bit-levels. This may reveal areas for optimization, including lower power and area in order to achieve similar or better results.

The focus of this research was comparing non-volatile memory to an equivalent volatile memory architecture. Most of the hardware specifications were the same. If the system hardware was taken into account, there could be other opportunities for optimization.

For example, the truly-random number generator is activated and the number is sent to a volatile memory space. Future developments could develop a single hardware unit which has both random number generation and memory which can be called once by the CPU to improve performance.

Other hardware changes could include investigating performance changes at slower or faster CPU clock speeds, lower or higher memory sizes, and the effects of parallelization to measure relative performance.

Additionally, this research focuses on just a single example of how memristors can be used to accelerate computing performance. There are many other operations which could be researched and the number of instructions can grow to lead to, on net, a large improvement in performance.

As memristor outputs are analog in nature, changing the output of each cell to support additional conductance levels can offer greater performance [34]. However, trade-

offs with security must also be considered and studied.

5.3 Security Verifications

The security implications of memristor processing units have not been studied thoroughly but will be necessary to develop sufficient trust among high-risk institutions. Future work should include a closer examination of this proposed TRNG to see whether it is more secure than the state-of-the-art of whether there could be novel attacks.

For example, external magnetic fields could be used to forcibly reset the memristor inputs [74]. A high temperature could generally affect the performance and lead to more predictability. There may be necessary mitigations in both hardware and software to prevent these kinds of attacks.

5.4 Hardware Integration

This research was not conducted using physical hardware. Future research should look at the physical limitations of memristors for in-memory computing, such as the upper bound for speed and endurance [46]. Adopting a different circuit design like a ring oscillator has been shown to theoretically improve generation speeds [67].

In this work, I only investigated STT-MRAM as a potential hardware choice. Future work could examine other kinds of non-volatile memory such as PCM to see if it would have faster switching times and allow for random numbers to generated faster.

5.5 NVM-Only Systems

The benefits of using memristors is clearest in the Internet of Things, but can grow over time as more improvements are developed. More research and development of in-memory computing can highlight additional performance changes. This may lead devices to forego NAND Flash and RAM entirely and use a hybrid system of memristor cells for

storing both volatile and non-volatile data.

Microcontrollers and other small mobile devices would benefit a lot, but these benefits could grow to include more powerful hardware. The development of NVM neural networks could accelerate a lot of modern applications while running offline, on-device. Although RAM by default is faster, additional benefits from processing in-memory can reduce the Von Neumann bottleneck and generally improve compute performance.

This kind of system would require many applications to be redesigned. Not only would they change the kind of memory, but the behavior would change. Closing an application would not necessarily reset the memory and a reboot would not necessarily fix a computer in a bad state. This would provide programmers with more choices in how their applications run across sessions.

Chapter 6

Conclusion

In this paper, I have developed an in-software simulation of a memristor processing unit and developed a TRNG using these memristors. This TRNG can generate numbers at a speed comparable or better than current architectures.

The RISC-V instruction set was amended to include three new commands relating to charging, discharging, and reading data to this processing unit. The RISC-V compiler and the Gem5 systems were updated in order to handle these new instructions and provide the CPU with random numbers.

A small program was written in C to call these new instructions to perform message scrambling. Compared to using a scrambling program using standard pseudo-random number generation, this new program was just as fast or faster.

This makes it a promising candidate for innovative applications which require truly random number generations. I believe this proposed TRNG can be a valuable tool for applications that require efficient security.

More generally, manufacturing of memristors will need greater scale and lower costs to make it broadly useful. We have seen more research and adoption of this technology in recent years and the field of non-volatile memory looks to continue growing in interest.

References

[1] X. Dong, A. Amirsoleimani, M. R. Azghadi, and R. Genov, "Wallax: A memristor-based gaussian random number generator," *Neurocomputing*, p. 126 933, 2023, ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2023.126933. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231223010561.

[2] D. W. Scott, "Box–muller transformation," *WIREs Computational Statistics*, vol. 3, no. 2, pp. 177–179, 2011. DOI: https://doi.org/10.1002/wics.148. eprint: https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/wics.148. [Online]. Available: https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/wics.148.

[3] G. Guerrer, "Rava: An open hardware true random number generator based on avalanche noise," *IEEE Access*, vol. 11, pp. 119 568–119 583, 2023. DOI: 10.1109/ACCESS.2023.3327325.

[4] B. Perach and S. Kvatinsky, "Stt-angie: Asynchronous true random number generator using stt-mtj," in *2019 Design, Automation, and Test in Europe Conference, and Exhibition (DATE)*, 2019, pp. 264–267. DOI: 10.23919/DATE.2019.8715257.

[5] L. Chua, "Memristor-the missing circuit element," *IEEE Transactions on Circuit Theory*, vol. 18, no. 5, pp. 507–519, 1971. DOI: 10.1109/TCT.1971.1083337.

[6] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, no. 7191, pp. 80–83, 2008. DOI: 10.1038/nature06932.

[7] Z. Biolek, D. Biolek, and V. Biolkova, "Computation of the area of memristor pinched hysteresis loop," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 59, no. 9, pp. 607–611, 2012. DOI: 10.1109/TCSII.2012.2208670.

[8] T. Kahale and D. Tannir, "Memristor modeling using the modified nodal analysis approach," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 4, pp. 1191–1195, 2022. DOI: 10.1109/TCAD.2021.3068098.

[9] V. Sverdlov, S. Fiorentini, J. Ender, W. Goes, R. L. de Orio, and S. Selberherr, "Emerging cmos compatible magnetic memories and logic," in *2020 IEEE Latin America Electron Devices Conference (LAEDC)*, 2020, pp. 1–4. DOI: 10.1109/LAEDC49063.2020.9073332.

[10] R. Bez, P. Cappelletti, G. Servalli, and A. Pirovano, "Phase change memories have taken the field," in *2013 5th IEEE International Memory Workshop*, 2013, pp. 13–16. DOI: 10.1109/IMW.2013.6582084.

[11] W. Chien *et al.*, "A novel self-converging write scheme for 2-bits/cell phase change memory for storage class memory (scm) application," in *2015 Symposium on VLSI Technology (VLSI Technology)*, 2015, T100–T101. DOI: 10.1109/VLSIT.2015.7223709.

[12] C. J. Yee, F. O. Hatem, T. N. Kumar, and H. A. F. Almurib, "Compact spice modeling of stt-mtj device," in *2015 IEEE Student Conference on Research and Development (SCOReD)*, 2015, pp. 625–628. DOI: 10.1109/SCORED.2015.7449413.

[13] X. Tian, Z. Bian, and H. Cai, "Towards near llc speed stt-mram sensing using reconfigurable clock trimming," in *2022 IEEE International Conference on Integrated Circuits, Technologies and Applications (ICTA)*, 2022, pp. 90–91. DOI: 10.1109/ICTA56932.2022.9963110.

[14] S. Kvatinsky, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Team: Threshold adaptive memristor model," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 60, no. 1, pp. 211–221, 2013. DOI: 10.1109/TCSI.2012.2215714.

[15] S. Kvatinsky, M. Ramadan, E. G. Friedman, and A. Kolodny, "Vteam: A general model for voltage-controlled memristors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 8, pp. 786–790, 2015. DOI: 10.1109/TCSII.2015.2433536.

[16] T. Luo *et al.*, "An fpga-based hardware emulator for neuromorphic chip with rram," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 2, pp. 438–450, 2020. DOI: 10.1109/TCAD.2018.2889670.

[17] J. Kim, Y. Song, K. Cho, H. Lee, H. Yoon, and E.-Y. Chung, "Stt-mram-based multicontext fpga for multithreading computing environment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 5, pp. 1330–1343, 2022. DOI: 10.1109/TCAD.2021.3091440.

[18] A. Aswani, R. Kumar, J. N. Tripathi, and A. James, "Performance of crossbar based long short term memory with aging memristors," in *2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2021, pp. 1–4. DOI: 10.1109/AICAS51828.2021.9458402.

[19] D. Chakraborty and S. K. Jha, "Automated synthesis of compact crossbars for sneak-path based in-memory computing," in *Design, Automation, and Test in Europe Conference and Exhibition (DATE), 2017.*

.

[20] K. Takeuchi, "Neuromorphic computing with computation-in-memory (cim)," in *2023 International VLSI Symposium on Technology, Systems and Applications (VLSI-TSA/VLSI-DAT)*, 2023, pp. 1–1. DOI: 10.1109/VLSI-TSA/VLSI-DAT57221.2023.10134289.

[21] C. D. Schuman, S. R. Kulkarni, M. Parsa, J. P. Mitchell, P. Date, and B. Kay, "Opportunities for neuromorphic computing algorithms and applications," *Nature Computational Science*, vol. 2, no. 1, Jan. 2022, ISSN: 2662-8457. DOI: 10.1038/s43588-021-00184-y. [Online]. Available: https://www.osti.gov/biblio/1881146.

[22] P. Joshi and H. Rahaman, "A comprehensive review on reram-based accelerators for deep learning," in *2023 International Symposium on Devices, Circuits and Systems (ISDCS)*, vol. 1, 2023, pp. 01–05. DOI: 10.1109/ISDCS58735.2023.10153519.

[23] P. A. Lázaro, I. J. Gallo, J. R. Aranda, A. del Barrio García, G. B. Juan, and F. J. Molinos, *Design and simulation of memristor-based neural networks*, 2023. arXiv: 2306.11678 [cs.ET].

[24] C. Lammie, W. Xiang, B. Linares-Barranco, and M. R. Azghadi, "Memtorch: An open-source simulation framework for memristive deep learning systems," *CoRR*, vol. abs/2004.10971, 2020. arXiv: 2004.10971. [Online]. Available: https://arxiv.org/abs/2004.10971.

[25] C. Lammie, W. Xiang, B. Linares-Barranco, and M. R. Azghadi, *coreylammie/MemTorch: Initial Release*, Apr. 2020. DOI: 10.5281/zenodo.3760695. [Online]. Available: https://doi.org/10.5281/zenodo.3760696.

[26] P. Yao *et al.*, "Fully hardware-implemented memristor convolutional neural network," *Nature*, vol. 577, no. 7792, pp. 641–646, Jan. 2020, ISSN: 1476-4687. DOI: 10.1038/s41586-020-1942-4. [Online]. Available: https://doi.org/10.1038/s41586-020-1942-4.

[27] W. Wan *et al.*, "A compute-in-memory chip based on resistive random-access memory," *Nature*, vol. 608, no. 7923, pp. 504–512, Aug. 2022, ISSN: 1476-4687. DOI: 10.1038/s41586-022-04992-8. [Online]. Available: https://doi.org/10.1038/s41586-022-04992-8.

[28] H. Ran, S. Wen, S. Wang, Y. Cao, P. Zhou, and T. Huang, "Memristor-based edge computing of shufflenetv2 for image classification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 8, pp. 1701–1710, 2021. DOI: 10.1109/TCAD.2020.3022970.

[29] M. Donato, L. Pentecost, D. Brooks, and G.-Y. Wei, "Memti: Optimizing on-chip nonvolatile storage for visual multitask inference at the edge," *IEEE Micro*, vol. 39, no. 6, pp. 73–81, 2019. DOI: 10.1109/MM.2019.2944782.

[30] D. Pala, I. Miro-Panades, and O. Sentieys, "Freezer: A specialized nvm backup controller for intermittently powered systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 8, pp. 1559–1572, 2021. DOI: 10.1109/TCAD.2020.3025063.

[31] S. Wood, D. Chakraborty, and J. Schmalzel, "Low power sensor fusion targeted for ai applications at the edge," in *2023 IEEE Sensors Applications Symposium (SAS)*, 2023, pp. 1–6. DOI: 10.1109/SAS58821.2023.10254113.

[32] J. Tan, S. Duan, L. Wang, and J. Yan, "Multi-gas sensing electronic nose using memristor-based in-memory computing," *IEEE Sensors Journal*, vol. 23, no. 22, pp. 28 526–28 534, 2023. DOI: 10.1109/JSEN.2023.3323943.

[33] V. Parmar, S. S. Sarwar, Z. Li, H.-H. S. Lee, B. D. Salvo, and M. Suri, "Exploring memory-oriented design optimization of edge ai hardware for extended reality applications," *IEEE Micro*, vol. 43, no. 6, pp. 40–49, 2023. DOI: 10.1109/MM.2023.3321249.

[34] L. Gao *et al.*, "Digital-to-analog and analog-to-digital conversion with metal oxide memristors for ultra-low power computing," in *2013 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, 2013, pp. 19–22. DOI: 10.1109/NanoArch.2013.6623031.

[35] S. Shin, K. Kim, and S.-M. Kang, "Memristor applications for programmable analog ics," *IEEE Transactions on Nanotechnology*, vol. 10, no. 2, pp. 266–274, 2011. DOI: 10.1109/TNANO.2009.2038610.

[36] I. Marković, M. Potrebić Ivaniš, and D. Tošić, "The dynamic tunability of memristor-based active filters," *Micromachines*, vol. 14, no. 11, p. 2064, Nov. 2023, ISSN: 2072-666X. DOI: 10.3390/mi14112064. [Online]. Available: http://dx.doi.org/10.3390/mi14112064.

[37] N. Felker *et al.*, "Building memsat: A cubesat for testing resistive memory," in *IEEE Sensors Application Symposium*, 2017.

[38] K. Konte, R. Trafford, and J. Schmalzel, "Implementing xeds for a cubesat communication subsystem," in *2018 IEEE Sensors Applications Symposium (SAS)*, 2018, pp. 1–5. DOI: 10.1109/SAS.2018.8336757.

[39] A. Memaripour and S. Swanson, "Breeze: User-level access to non-volatile main memories for legacy software," in *2018 IEEE 36th International Conference on Computer Design, 2018.*

.

[40] M. M. H. Sumon, M. L. Ali, and M. S. Sadi, "An efficient triple-error correction method for memristor-based memory systems," in *2022 12th International Conference on Electrical and Computer Engineering (ICECE)*, 2022, pp. 380–383. DOI: 10.1109/ICECE57408.2022.10088479.

[41] C. Ye, Y. Xu, X. Shen, X. Liao, H. Jin, and Y. Solihin, "Supporting legacy libraries on non-volatile memory: A user-transparent approach," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 443–455. DOI: 10.1109/ISCA52012.2021.00042.

[42] A. Agrawal *et al.*, "Revisiting stochastic computing in the era of nanoscale nonvolatile technologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 12, pp. 2481–2494, 2020. DOI: 10.1109/TVLSI.2020.2991679.

[43] S. Abreu, *Concepts and paradigms for neuromorphic programming*, 2023. arXiv: 2310.18260 [cs.NE].

[44] S. T. Sliper *et al.*, "Pragmatic memory-system support for intermittent computing using emerging nonvolatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 1, pp. 95–108, 2023. DOI: 10.1109/TCAD.2022.3168263.

[45] Unknown, *Dagstuhl: Formal methods for correct persistent programming – alastair reid – researcher at intel*. [Online]. Available: https://alastairreid.github.io/dagstuhl/.

[46] H. Zhou, J. Chen, J. Li, L. Yang, Y. Li, and X. Miao, "Bring memristive in-memory computing into general-purpose machine learning: A perspective," *APL Machine Learning*, vol. 1, no. 4, p. 040 901, Oct. 2023, ISSN: 2770-9019. DOI: 10.1063/5.0167743. eprint: https://pubs.aip.org/aip/aml/article-pdf/doi/10.1063/5.0167743/18198793/040901_1_5.0167743.pdf. [Online]. Available: https://doi.org/10.1063/5.0167743.

[47] Y.-J. Wu, C.-Y. Kuo, and L.-P. Chang, "Invmfs: An efficient file system for nvram-based intermittent computing devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 3638–3649, 2022. DOI: 10.1109/TCAD.2022.3197485.

[48] M. Rao *et al.*, "Thousands of conductance levels in memristors integrated on cmos," *Nature*, vol. 615, no. 7954, pp. 823–829, Mar. 2023, ISSN: 1476-4687. DOI: 10.1038/s41586-023-05759-5. [Online]. Available: https://doi.org/10.1038/s41586-023-05759-5.

[49] N. Felker, S. Shin, and S.-M. S. Kang, "Memos for performance analysis of future memristive memory-based computer architectures and simulation," *Journal of Open Source Developments*, vol. 5, no. 2, 2018.

[50] R. F. Freitas and W. W. Wilcke, "Storage-class memory: The next storage system technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 439–447, 2008. DOI: 10.1147/rd.524.0439.

[51] W. Banerjee, "Challenges and applications of emerging nonvolatile memory devices," *Electronics*, vol. 9, no. 6, 2020, ISSN: 2079-9292. DOI: 10.3390/electronics9061029. [Online]. Available: https://www.mdpi.com/2079-9292/9/6/1029.

[52] A. Haj-Ali, R. Ben-Hur, N. Wald, R. Ronen, and S. Kvatinsky, "Not in name alone: A memristive memory processing unit for real in-memory processing," *IEEE Micro*, vol. 38, no. 5, pp. 13–21, 2018. DOI: 10.1109/MM.2018.053631137.

[53] R. Ben Hur, N. Wald, N. Talati, and S. Kvatinsky, "Simple magic: Synthesis and in-memory mapping of logic execution for memristor-aided logic," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 225–232. DOI: 10.1109/ICCAD.2017.8203782.

[54] L. Song, F. Chen, H. Li, and Y. Chen, "Refloat: Low-cost floating-point processing in reram for accelerating iterative linear solvers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23, Denver, CO, USA: Association for Computing Machinery, 2023, ISBN: 9798400701092. DOI: 10.1145/3581784.3607077. [Online]. Available: https://doi.org/10.1145/3581784.3607077.

[55] V. Costan, I. Lebedev, and S. Devadas. 2017.

[56] T. Roche, V. Lomné, C. Mutschler, and L. Imbert, "A side journey to titan," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, pp. 231–248, ISBN: 978-1-939133-24-3. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/roche.

[57] S. Ghorbani Lyastani, M. Schilling, M. Neumayr, M. Backes, and S. Bugiel, "Is fido2 the kingslayer of user authentication? a comparative usability study of fido2 passwordless authentication," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 268–285. DOI: 10.1109/SP40000.2020.00047.

[58] T. Ngo and A. Williams, "Entropy lost: Nintendo's not-so-random sequence of 32, 767 bits," in *Proceedings of the 17th International Conference on the Foundations of Digital Games*, ser. FDG '22, Athens, Greece: Association for Computing Machinery, 2022, ISBN: 9781450397957. DOI: 10.1145/3555858.3563265. [Online]. Available: https://doi.org/10.1145/3555858.3563265.

[59] Unknown, *Sp 800-90a rev. 1, recommendation for random number generation using deterministic random bit generators*. [Online]. Available: https://csrc.nist.gov/pubs/sp/800/90/a/r1/final.

[60] *CVE-2023-39910.* Available from MITRE, CVE-ID CVE-2023-39910. Aug. 2023. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-39910.

[61] K. Ryan, G. A. Sullivan, K. He, and N. Heninger, "Passive ssh key compromise via lattices," in *2023 ACM Computer and Communications Security Conference (ACM CCS)*, Nov. 2023. DOI: 10.1145/3576915.3616629.

[62] L. Dorrendorf, Z. Gutterman, and B. Pinkas, "Cryptanalysis of the random number generator of the windows operating system," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, Nov. 2009, ISSN: 1094-9224. DOI: 10.1145/1609956.1609966. [Online]. Available: https://doi.org/10.1145/1609956.1609966.

[63] S. N. Cohney, M. D. Green, and N. Heninger, "Practical state recovery attacks against legacy rng implementations," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, Toronto, Canada: Association for Computing Machinery, 2018, pp. 265–280, ISBN: 9781450356930. DOI: 10.1145/3243734.3243756. [Online]. Available: https://doi.org/10.1145/3243734.3243756.

[64] C. Yao, Y. Liu, X. Wei, G. Wang, and F. Gao, "Backscatter technologies and the future of internet of things: Challenges and opportunities," *Intelligent and Converged Networks*, vol. 1, no. 2, pp. 170–180, 2020. DOI: 10.23919/ICN.2020.0013.

[65] A. Harit, A. Ezzati, and R. Elharti, "Internet of things security: Challenges and perspectives," in *Proceedings of the Second International Conference on Internet of Things, Data and Cloud Computing*, ser. ICC '17, Cambridge, United Kingdom: Association for Computing Machinery, 2017, ISBN: 9781450347747. DOI: 10.1145/3018896.3056784. [Online]. Available: https://doi.org/10.1145/3018896.3056784.

[66] B. Williams, R. E. Hiromoto, and A. Carlson, "A design for a cryptographically secure pseudo random number generator," in *2019 10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, vol. 2, 2019, pp. 864–869. DOI: 10.1109/IDAACS.2019.8924431.

[67] D. Arumí, S. Manich, A. Gómez-Pau, R. Rodríguez-Montañés, M. B. González, and F. Campabadal, "True random number generator based on rram-bias current starved ring oscillator," *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, vol. 9, no. 2, pp. 92–98, 2023. DOI: 10.1109/JXCDC.2023.3320056.

[68] N. Bigdeli, Y. Farid, and K. Afshar, "A robust hybrid method for image encryption based on hopfield neural network," *Comput. Electr. Eng.*, vol. 38, no. 2, pp. 356–369, Mar. 2012, ISSN: 0045-7906. DOI: 10.1016/j.compeleceng.2011.11.019. [Online]. Available: https://doi.org/10.1016/j.compeleceng.2011.11.019.

[69] H. Lin, C. Wang, C. Xu, X. Zhang, and H. H. C. Iu, "A memristive synapse control method to generate diversified multistructure chaotic attractors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 3, pp. 942–955, 2023. DOI: 10.1109/TCAD.2022.3186516.

[70] P. Das and S. Mandal, "A physical memristor-based chaotic system and its application in colour image encryption scheme," *Physica Scripta*, vol. 98, no. 11, p. 115 252, Oct. 2023. DOI: 10.1088/1402-4896/ad033f. [Online]. Available: https://dx.doi.org/10.1088/1402-4896/ad033f.

[71] A. Chen and Y. Zhang, "A novel pseudo-random number assisted fast image encryption algorithm," *Multimedia Tools and Applications*, Oct. 2023, ISSN: 1573-7721. DOI: 10.1007/s11042-023-17209-5. [Online]. Available: https://doi.org/10.1007/s11042-023-17209-5.

[72] Q. Dong *et al.*, "A 1mb 28nm stt-mram with 2.8ns read access time at 1.2v vdd using single-cap offset-cancelled sense amplifier and in-situ self-write-termination," in *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, 2018, pp. 480–482. DOI: 10.1109/ISSCC.2018.8310393.

[73] Unknown, *Github - riscv-collab/riscv-gnu-toolchain: Gnu toolchain for risc-v, including gcc*. [Online]. Available: https://github.com/riscv-collab/riscv-gnu-toolchain.

[74] L. Azriel and S. Kvatinsky, "Towards a memristive hardware secure hash function (memhash)," in *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2017, pp. 51–55. DOI: 10.1109/HST.2017.7951797.

[75] G. development team, *Gem5*. [Online]. Available: https://www.gem5.org/.

[76] Unknown, *Unknown*. [Online]. Available: https://www.gem5.org/documentation/general_docs/architecture_support/isa_parser/.

[77] A. Rukhin *et al.*, *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. NIST: National Institute of Standards and Technology, Apr. 2010.

[78] S. Song, P. Huang, W. Shen, L. Liu, and J. Kang, "A 3.3-mbit/s true random number generator based on resistive random access memory," *Science China Information Sciences*, vol. 66, no. 11, p. 219 402, Oct. 2023, ISSN: 1869-1919. DOI: 10.1007/s11432-022-3640-0. [Online]. Available: https://doi.org/10.1007/s11432-022-3640-0.

[79] A. Marsh, *John Deere and the Birth of Precision Agriculture — spectrum.ieee.org*, https://spectrum.ieee.org/john-deere-and-the-birth-of-precision-agriculture, [Accessed 02-12-2023].

[80] C.-S. Park and J.-H. Lee, "Security bootstrapping for secure join and binding on the ieee 802.15.4-based lowpan," *IEEE Internet of Things Journal*, vol. 4, no. 4, pp. 996–1005, 2017. DOI: 10.1109/JIOT.2017.2709757.

[81] J. P. Tuohy, *Your Roomba can now be a security camera — theverge.com*, https://www.theverge.com/2023/4/17/23686553/irobot-roomba-j7-security-camera-beta-feature, [Accessed 02-12-2023].

Appendix A

Code for Gem5 Memristor TRNG

```
static float C_HIGH = 20;
static float C_LOW = 0;

class Capacitor
{
    public:
        uint32_t farads;
        float coulombs;
        Capacitor() {
            // Default
            this->farads = 50;
        }
        Capacitor(uint32_t farads) {
            this->farads = farads;
        }
        void setCapacitance(float coulombs) {
            if (coulombs < C_LOW) {
                this->coulombs = C_LOW;
            } else if (coulombs > C_HIGH) {
                this->coulombs = C_HIGH;
            } else {
                this->coulombs = coulombs;
            }
        }
        void charge(float volts) {
            // V = Q/C
```

```
            // V*C = Q
            this->setCapacitance(volts * this->farads);
        }
        float getVoltage() {
            // V = Q/C
            return this->coulombs / this->farads;
        }
};

static uint32_t R_HIGH = 100000;
static uint32_t R_LOW = 1000;

class SttMtj
{
    public:
        uint32_t memristance;
        uint32_t resistance;
        SttMtj() {
            // Default
            this->memristance = 50;
        }
        SttMtj(uint32_t memristance) {
            this->memristance = memristance;
        }
        void setResistance(uint32_t ohms) {
            if (ohms < R_LOW) {
                this->resistance = R_LOW;
            } else if (ohms > R_HIGH) {
                this->resistance = R_HIGH;
            } else {
                this->resistance = ohms;
            }
        }
```

```
        void applyWrite(float volts) {
            if (volts > 0) {
                // Writing
                uint32_t prob = rand() % 100;
                if (prob < this->memristance) {
                    this->setResistance(R_HIGH);
                } else {
                    this->setResistance(R_LOW);
                }
            }
        }
        bool read() {
            uint32_t middleLine = (R_HIGH - R_LOW) / 2;
            if (this->resistance > middleLine) {
                return true;
            }
            return false;
        }
};

static uint32_t memristances[] = {
    30, 40, 87, 70, 24, 13, 92, 51, //  8
     1, 68, 84, 22, 35, 18, 43, 77, // 16
    20, 30, 40, 19, 51, 62, 98, 44, // 24
    18, 64, 37, 59, 21, 45, 82, 93, // 32
};

#define MEMSIZE 32

class SttMramTrng
{
    public:
        Capacitor cap;
```

```
        SttMtj row[MEMSIZE];
        uint32_t length;
        SttMramTrng(uint32_t n) {
            this->cap = Capacitor(50);
            this->length = n;
            for (uint32_t i = 0; i < n; i++) {
                row[i] = SttMtj(memristances[i]);
            }
        }
        void reset(uint32_t seed) {
            srand(seed);
            this->cap.charge(5.0); // 5v
        }
        void enable() {
            this->cap.charge(0.0); // Mimic discharge
            for (uint32_t i = 0; i < this->length; i++) {
                this->row[i].applyWrite(5.0); // 5v
            }
        }
        uint32_t read() {
            uint32_t random = 0;
            for (uint32_t i = 0; i < this->length; i++) {
                // Read each cell in a row to get final out.
                random = random << 1;
                if (this->row[i].read()) {
                    random |= 1;
                }
            }
            return random;
        }
};

static SttMramTrng __sttMramTrng(MEMSIZE);
```

```
constexpr SttMramTrng* _sttMramTrng = &__sttMramTrng;
```

Appendix B

Generate N Random Numbers Program

```
#define MEM_DDR4 1
#define MEM_MTJ  2

#if MEM_KIND == MEM_MTJ
uint32_t truerandom(uint32_t* rng, uint32_t seed) {
    uint32_t p1 = seed;
    uint32_t p2 = 0;
    asm volatile("trng.reset %0, %1, %2\n":"=r"(*rng):"r"(p1),"r"(p2):);
    asm volatile("trng.enable %0, %1, ...
        %2\n":"=r"(*rng):"r"(p1),"r"(p2):);
    asm volatile("trng.read %0, %1, %2\n":"=r"(*rng):"r"(p1),"r"(p2):);
}
#endif

#define ITERATIONS 1024
int main(void)
{
    for (uint32_t i = 0; i < ITERATIONS; i++) {
        #if MEM_KIND == MEM_DDR4
        srand(i * 100000 + i);
        uint32_t rng = rand();
        #endif

        #if MEM_KIND == MEM_MTJ
        uint32_t rng;
        truerandom(&rng, i * 100000 + i);
```

```
        #endif
        printf("%i\n",rng);
    }
    return 0;
}
```

Appendix C

Scramble Program

```
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MEM_DDR4 1
#define MEM_MTJ  2

char scrambleChars[] = {
    'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',  //  8
    'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P',  // 16
    'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',  // 24
    'Y', 'Z', '0', '1', '2', '3', '4', '5',  // 32
    '6', '7', '8', '9', ' ', '-', '_', '=',  // 40
    '!', '?', '.', ',', '<', '>', '/', '@',  // 48
    '#', '$', '%', '^', '&', '*', '(', ')',  // 56
    '`', '¬', '[', ']', '{', '}', '|', ':',  // 64
};

#if MEM_KIND == MEM_DDR4
void scramble(char msg[], char output[], uint32_t length) {
    for (uint32_t i = 0; i < length; i++) {
        srand(i * 100000 + msg[i]);
        uint32_t randIndex = rand();
        output[i] = scrambleChars[randIndex % 64];
```

```
        printf("Scramble char %i: %c -%i-> %c\n", i, msg[i], ...
            randIndex, output[i]);
    }
}
#endif

#if MEM_KIND == MEM_MTJ
uint32_t truerandom(uint32_t* rng, uint32_t seed) {
    uint32_t p1 = seed;
    uint32_t p2 = 0;
    asm volatile("trng.reset %0, %1, %2\n":"=r"(*rng):"r"(p1),"r"(p2):);
    asm volatile("trng.enable %0, %1, ...
        %2\n":"=r"(*rng):"r"(p1),"r"(p2):);
    asm volatile("trng.read %0, %1, %2\n":"=r"(*rng):"r"(p1),"r"(p2):);
}

void scramble(char msg[], char output[], uint32_t length) {
    for (uint32_t i = 0; i < length; i++) {
        uint32_t randIndex;
        truerandom(&randIndex, i * 100000 + msg[i]);
        output[i] = scrambleChars[randIndex % 64];
        printf("Scramble char %i: %c -%i-> %c\n", i, msg[i], ...
            randIndex, output[i]);
    }
}
#endif

void getSecret(char msg[], char output[], uint32_t length) {
    #if MEM_KIND == MEM_DDR4
    char hashedSecret[] = "}0TPSTKFJ&G";
    #endif
    #if MEM_KIND == MEM_MTJ
    char hashedSecret[] = "BN,?9?==$9A";
```

```
    #endif
    char pwd[] = "MYPWD";
    char noMatch[] = "ERROR";
    if (strcmp(msg, hashedSecret) == 0) {
        for (uint32_t i = 0; i < length; i++) {
            output[i] = pwd[i];
        }
    } else {
        // No match
        for (uint32_t i = 0; i < length; i++) {
            output[i] = noMatch[i];
        }
    }
}

#define ITERATIONS 5
int main(void)
{
  for (uint32_t i = 0; i < ITERATIONS; i++) {
    char in[] = "Hello World";
    char out[11];
    scramble(in, out, 11);
    printf("Scramble msg %s\n", out);
  }

  char userInputPwd[] = "Hello World";
  char hashInputPwd[11];
  scramble(userInputPwd, hashInputPwd, 11);
  char outputPwd[5];
  getSecret(hashInputPwd, outputPwd, 5);
  printf("Scramble msg %s\n", hashInputPwd);
  printf("Secure enclave returned: %s\n", outputPwd);
```

```
  char userInputPwd2[] = "BadPassword";
  char hashInputPwd2[11];
  scramble(userInputPwd2, hashInputPwd2, 11);
  char outputPwd2[5];
  getSecret(hashInputPwd2, outputPwd2, 5);
  printf("Scramble msg %s\n", hashInputPwd2);
  printf("Secure enclave returned: %s\n", outputPwd2);
  return 0;
}
```

Appendix D

Measure Zeros and Ones Program

```
const cpu = [ /* Insert numbers here */ ]
const mtj = [ /* Insert numbers here */ ]
const wind = [ /* Insert numbers here */ ]
const cpuToBin = cpu.map(num => num.toString(2).padStart(32, '0'))
const mtjToBin = mtj.map(num => num.toString(2).padStart(32, '0'))
const winToBin = wind.map(num => num.toString(2).padStart(32, '0'))

const countVal = (str, match) => {
  return str.split('').filter(char => char === match).length
}

console.log('CPU:')
const cpuOnes = cpuToBin.map(b => countVal(b, '1'))
const cpuZeros = cpuToBin.map(b => countVal(b, '0'))
console.log('> 1:')
cpuOnes.forEach(n => console.log(n))
console.log('> 0:', cpuZeros[1])
cpuZeros.forEach(n => console.log(n))

console.log('MTJ:')
const mtjOnes = mtjToBin.map(b => countVal(b, '1'))
const mtjZeros = mtjToBin.map(b => countVal(b, '0'))
console.log('> 1:')
mtjOnes.forEach(n => console.log(n))
console.log('> 0:')
mtjZeros.forEach(n => console.log(n))
```

```
console.log('Windows:')
const winOnes = winToBin.map(b => countVal(b, '1'))
const winZeros = winToBin.map(b => countVal(b, '0'))
console.log('> 1:')
winOnes.forEach(n => console.log(n))
console.log('> 0:')
winOnes.forEach(n => console.log(n))
```