D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput

Jeremie S. Kim[‡§] Minesh Patel[§] Hasan Hassan[§] Lois Orosa[§] Onur Mutlu[§‡]
[‡]Carnegie Mellon University [§]ETH Zürich

We propose a new DRAM-based true random number generator (TRNG) that leverages DRAM cells as an entropy source. The key idea is to intentionally violate the DRAM access timing parameters and use the resulting errors as the source of randomness. Our technique specifically decreases the DRAM row activation latency (timing parameter t_{RCD}) below manufacturer-recommended specifications, to induce read errors, or activation failures, that exhibit true random behavior. We then aggregate the resulting data from multiple cells to obtain a TRNG capable of providing a high throughput of random numbers at low latency.

To demonstrate that our TRNG design is viable using commodity DRAM chips, we rigorously characterize the behavior of activation failures in 282 state-of-the-art LPDDR4 devices from three major DRAM manufacturers. We verify our observations using four additional DDR3 DRAM devices from the same manufacturers. Our results show that many cells in each device produce random data that remains robust over both time and temperature variation. We use our observations to develop D-RaNGe, a methodology for extracting true random numbers from commodity DRAM devices with high throughput and low latency by deliberately violating the read access timing parameters. We evaluate the quality of our TRNG using the commonly-used NIST statistical test suite for randomness and find that D-RaNGe: 1) successfully passes each test, and 2) generates true random numbers with over two orders of magnitude higher throughput than the previous highest-throughput DRAM-based TRNG.

1. Introduction

Random number generators (RNGs) are critical components in many different applications, including cryptography, scientific simulation, industrial testing, and recreational entertainment [13,15,31,37,47,69,80,82,95,121,135,142,152,162]. These applications require a mechanism capable of rapidly generating random numbers across a wide variety of operating conditions (e.g., temperature/voltage fluctuations, manufacturing variations, malicious external attacks) [158]. In particular, for modern cryptographic applications, a random (i.e., completely unpredictable) number generator is critical to prevent information leakage to a potential adversary [31, 37, 47, 69, 79, 80, 82, 152, 162].

Random number generators can be broadly classified into two categories [32, 78, 145, 148]: 1) *pseudo-random number generators (PRNGs)* [18, 98, 100, 102, 133], which deterministically generate numbers starting from a *seed value* with the goal of approximating a true random sequence, and 2) *true random number generators (TRNGs)* [6, 16, 22, 23, 24, 33, 36, 47, 50, 55, 56, 57, 65, 77, 83, 96, 101, 111, 116, 119, 141, 143, 144, 146, 149, 151, 153, 155, 158], which generate random numbers based on sampling non-deterministic random variables inherent in various physical phenomena (e.g., electrical noise, atmospheric noise, clock jitter, Brownian motion).

PRNGs are popular due to their flexibility, low cost, and fast pseudo-random number generation time [24], but their output is *fully determined* by the starting seed value. This means that the output of a PRNG may be predictable given complete information about its operation. Therefore, a PRNG falls short for applications that require high-entropy values [31, 35, 152]. In contrast, because a TRNG mechanism relies on sampling entropy inherent in *non-deterministic* physical phenomena, the output of a TRNG is *fully unpredictable* even when complete information about the underlying mechanism is available [79].

Based on analysis done by prior work on TRNG design [64, 79, 124], we argue that an *effective* TRNG must: 1) produce truly random (i.e., completely unpredictable) numbers, 2) provide a high throughput of random numbers at low latency, and 3) be practically implementable at low cost. Many prior works study different methods of generating true random numbers that can be implemented using CMOS devices [6, 16, 22, 23, 24, 33, 36, 47, 50, 55, 56, 57, 65, 77, 83, 96, 101, 111, 116, 119, 141, 143, 144, 146, 149, 151, 153, 155, 158]. We provide a thorough discussion of these past works in Section 9. Unfortunately, most of these proposals fail to satisfy all of the properties of an effective TRNG because they either require specialized hardware to implement (e.g., free-running oscillators [6, 158], metastable circuitry [16, 22, 101, 146]) or are unable to sustain continuous high-throughput operation on the order of Mb/s (e.g., memory startup values [39, 55, 56, 144, 151], memory data retention failures [65, 141]). These limitations preclude the widespread adoption of such TRNGs, thereby limiting the overall impact of these proposals.

Commodity DRAM chips offer a promising substrate to overcome these limitations due to three major reasons. First, DRAM operation is highly sensitive to changes in access timing, which means that we can easily induce failures by manipulating manufacturer-recommended DRAM access timing parameters. These failures have been shown to exhibit non-determinism [27, 66, 71, 72, 84, 87, 109, 112, 117, 157] and therefore they may be exploitable for true random number generation. Second, commodity DRAM devices already provide an interface capable of transferring data continuously with high throughput in order to support a high-performance TRNG. Third, DRAM devices are already prevalently in use throughout modern computing systems, ranging from simple microcontrollers to sophisticated supercomputers.

Our goal in this paper is to design a TRNG that:

1. is implementable on commodity DRAM devices today
2. is fully non-deterministic (i.e., it is impossible to predict the next output even with complete information about the underlying mechanism)
3. provides continuous (i.e., constant rate), high-throughput random values at low latency
4. provides random values while minimally affecting concurrently-running applications

Meeting these four goals would enable a TRNG design that is suitable for applications requiring high-throughput true random number generation in commodity devices today.

Prior approaches to DRAM-based TRNG design successfully use DRAM data retention failures [50, 65, 141], DRAM startup values [39, 144], and non-determinism in DRAM com-

mand scheduling [116] to generate true random numbers. Unfortunately, these approaches do not fully satisfy our four goals because they either do not exploit a fundamentally non-deterministic entropy source (e.g., DRAM command scheduling [116]) or are too slow for continuous high-throughput operation (e.g., DRAM data retention failures [50, 65, 141], DRAM startup values [39, 144]). Section 8 provides a detailed comparative analysis of these prior works.

In this paper, we propose a new way to leverage DRAM cells as an entropy source for true random number generation by intentionally violating the access timing parameters and using the resulting errors as the source of randomness. Our technique specifically extracts randomness from *activation failures*, i.e., DRAM errors caused by intentionally decreasing the row activation latency (timing parameter t_{RCD}) below manufacturer-recommended specifications. Our proposal is based on **two key observations**:

1. Reading certain DRAM cells with a reduced activation latency returns *true random* values.
2. An activation failure can be induced very quickly (i.e., *even faster* than a normal DRAM row activation).

Based on these key observations, we propose D-RaNGe, a new methodology for extracting true random numbers from commodity DRAM devices with high throughput. D-RaNGe consists of two steps: 1) identifying specific DRAM cells that are vulnerable to activation failures using a *low-latency* profiling step and 2) generating a continuous stream (i.e., constant rate) of random numbers by repeatedly inducing activation failures in the previously-identified vulnerable cells. D-RaNGe runs entirely in software and is capable of immediately running on any commodity system that provides the ability to manipulate DRAM timing parameters within the memory controller [7, 8]. For most other devices, a simple software API must be exposed without any hardware changes to the commodity DRAM device (e.g., similarly to SoftMC [52, 132]), which makes D-RaNGe suitable for implementation on most existing systems today.

In order to demonstrate D-RaNGe's effectiveness, we perform a rigorous experimental characterization of activation failures using 282 state-of-the-art LPDDR4 [63] DRAM devices from three major DRAM manufacturers. We also verify our observations using four additional DDR3 [62] DRAM devices from a single manufacturer. Using the standard NIST statistical test suite for randomness [122], we show that D-RaNGe is able to maintain high-quality true random number generation both over 15 days of testing and across the entire reliable testing temperature range of our infrastructure (55°C-70°C). Our results show that D-RaNGe's maximum (average) throughput is 717.4Mb/s (435.7Mb/s) using four LPDDR4 DRAM channels, which is over two orders of magnitude higher than that of the best prior DRAM-based TRNG.

We make the following **key contributions**:

1. We introduce D-RaNGe, a new methodology for extracting true random numbers from a commodity DRAM device at high throughput and low latency. The key idea of D-RaNGe is to use DRAM cells as entropy sources to generate true random numbers by accessing them with a latency that is lower than manufacturer-recommended specifications.
2. Using experimental data from 282 state-of-the-art LPDDR4 DRAM devices from three major DRAM manufacturers, we present a rigorous characterization of randomness in errors induced by accessing DRAM with low latency. Our analysis demonstrates that D-RaNGe is able to maintain high-quality random number generation both over 15 days of testing and across the entire reliable testing temperature range of our infrastructure (55°C-70°C). We verify our observations from this study with prior works' observations on DDR3 DRAM devices [27, 71, 84, 87]. Furthermore, we experimentally demonstrate on four DDR3 DRAM devices, from a single manufacturer, that D-RaNGe is suitable for implementation in a wide range of commodity DRAM devices.
3. We evaluate the quality of D-RaNGe's output bitstream using the standard NIST statistical test suite for randomness [122] and find that it successfully passes every test. We also compare D-RaNGe's performance to four previously proposed DRAM-based TRNG designs (Section 8) and show that D-RaNGe outperforms the best prior DRAM-based TRNG design by over two orders of magnitude in terms of maximum and average throughput.

2. Background

We provide the necessary background on DRAM and true random number generation that is required to understand our idea of true random number generation using the inherent properties of DRAM.

2.1. Dynamic Random Access Memory (DRAM)

We briefly describe DRAM organization and basics. We refer the reader to past works [17, 26, 27, 28, 29, 30, 46, 51, 52, 66, 67, 68, 71, 72, 73, 75, 76, 84, 85, 87, 88, 89, 91, 92, 105, 112, 117, 125, 127, 128, 161, 161] for more detail.

2.1.1. DRAM System Organization. In a typical system configuration, a CPU chip includes a set of memory controllers, where each memory controller interfaces with a DRAM channel to perform read and write operations. As we show in Figure 1 (left), a DRAM channel has its own I/O bus and operates independently of other channels in the system. To achieve high memory capacity, a channel can host multiple DRAM modules by sharing the I/O bus between the modules. A DRAM module implements a single or multiple DRAM ranks. Command and data transfers are serialized between ranks in the same channel due to the shared I/O bus. A DRAM rank consists of multiple DRAM chips that operate in lock-step, i.e., all chips simultaneously perform the same operation, but they do so on different bits. The number of DRAM chips per rank depends on the data bus width of the DRAM chips and the channel width. For example, a typical system has a 64-bit wide DRAM channel. Thus, four 16-bit or eight 8-bit DRAM chips are needed to build a DRAM rank.

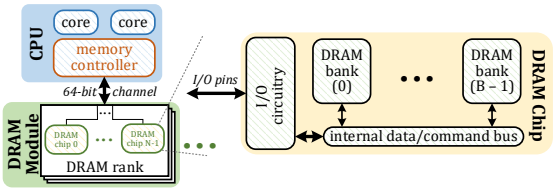


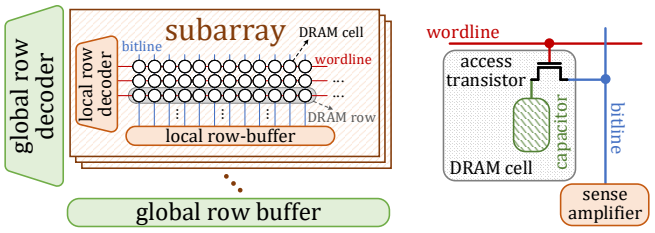
Figure 1: A typical DRAM-based system [71].

2.1.2. DRAM Chip Organization. At a high-level, a DRAM chip consists of billions of DRAM cells that are hierarchically organized to maximize storage density and performance. We describe each level of the hierarchy of a modern DRAM chip.

A modern DRAM chip is composed of multiple DRAM banks (shown in Figure 1, right). The chip communicates with the memory controller through the *I/O circuitry*. The I/O circuitry is connected to the *internal command and data bus* that is shared among all banks in the chip.

Figure 2a illustrates the organization of a DRAM bank. In a bank, the *global row decoder* partially decodes the address of the accessed *DRAM row* to select the corresponding *DRAM subarray*. A DRAM subarray is a 2D array of DRAM cells, where cells are horizontally organized into multiple DRAM

rows. A DRAM row is a set of DRAM cells that share a wire called the *wordline*, which the *local row decoder* of the subarray drives after fully decoding the row address. In a subarray, a column of cells shares a wire, referred to as the *bitline*, that connects the column of cells to a *sense amplifier*. The sense amplifier is the circuitry used to read and modify the data of a DRAM cell. The row of sense amplifiers in the subarray is referred to as the *local row-buffer*. To access a DRAM cell, the corresponding DRAM row first needs to be copied into the local row-buffer, which connects to the internal I/O bus via the *global row-buffer*.



(a) DRAM bank. (b) DRAM cell.

Figure 2: DRAM bank and cell architecture [71].

Figure 2b illustrates a DRAM cell, which is composed of a *storage capacitor* and *access transistor*. A DRAM cell stores a single bit of information based on the charge level of the capacitor. The data stored in the cell is interpreted as a "1" or "0" depending on whether the charge stored in the cell is above or below a certain threshold. Unfortunately, the capacitor and the access transistor are not ideal circuit components and have *charge leakage paths*. Thus, to ensure that the cell does not leak charge to the point where the bit stored in the cell flips, the cell needs to be periodically *refreshed* to fully restore its original charge.

2.1.3. DRAM Commands. The memory controller issues a set of DRAM commands to access data in the DRAM chip. To perform a read or write operation, the memory controller first needs to *open* a row, i.e., copy the data of the cells in the row to the row-buffer. To open a row, the memory controller issues an *activate (ACT)* command to a bank by specifying the address of the row to open. The memory controller can issue *ACT* commands to different banks in consecutive DRAM bus cycles to operate on *multiple banks in parallel*. After opening a row in a bank, the memory controller issues either a *READ* or a *WRITE* command to read or write a DRAM word (which is typically equal to 64 bytes) within the open row. An open row can serve multiple *READ* and *WRITE* requests without incurring precharge and activation delays. A DRAM row typically contains 4-8 KiBs of data. To access data from another DRAM row in the same bank, the memory controller must first close the currently open row by issuing a *precharge (PRE)* command. The memory controller also periodically issues *refresh (REF)* commands to prevent data loss due to charge leakage.

2.1.4. DRAM Cell Operation. We describe DRAM operation by explaining the steps involved in reading data from a DRAM cell.[1] The memory controller initiates each step by issuing a DRAM command. Each step takes a certain amount of time to complete, and thus, a DRAM command is typically associated with one or more timing constraints known as *timing parameters*. It is the responsibility of the memory controller to satisfy these timing parameters in order to ensure *correct* DRAM operation.

[1]Although we focus only on reading data, steps involved in a write operation are similar.

In Figure 3, we show how the state of a DRAM cell changes during the steps involved in a read operation. Each DRAM cell diagram corresponds to the state of the cell at exactly the tick mark on the time axis. Each command (shown in purple boxes below the time axis) is issued by the memory controller at the corresponding tick mark. Initially, the cell is in a *precharged* state ①. When precharged, the capacitor of the cell is disconnected from the bitline since the wordline is not asserted and thus the access transistor is off. The bitline voltage is stable at $\frac{V_{dd}}{2}$ and is ready to be perturbed towards the voltage level of the cell capacitor upon enabling the access transistor.

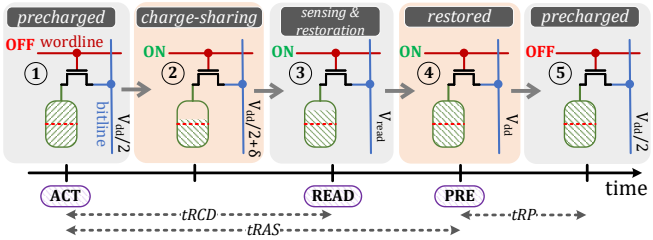


Figure 3: Command sequence for reading data from DRAM and the state of a DRAM cell during each related step.

To read data from a cell, the memory controller first needs to perform *row activation* by issuing an *ACT* command. During row activation (②), the row decoder asserts the wordline that connects the storage capacitor of the cell to the bitline by enabling the access transistor. At this point, the capacitor charge perturbs the bitline via the *charge sharing* process. Charge sharing continues until the capacitor and bitline voltages reach an equal value of $\frac{V_{dd}}{2} + \delta$. After charge sharing (③), the sense amplifier begins driving the bitline towards either V_{dd} or $0V$ depending on the direction of the perturbation in the charge sharing step. This step, which amplifies the voltage level on the bitline as well as the cell is called *charge restoration*. Although charge restoration continues until the original capacitor charge is fully replenished (④), the memory controller can issue a *READ* command to safely read data from the activated row before the capacitor charge is fully replenished. A *READ* command can reliably be issued when the bitline voltage reaches the voltage level V_{read}. To ensure that the read occurs after the bitline reaches V_{read}, the memory controller inserts a time interval t_{RCD} between the *ACT* and *READ* commands. It is the responsibility of the DRAM manufacturer to ensure that their DRAM chip operates safely as long as the memory controller obeys the t_{RCD} timing parameter, which is defined in the DRAM standard [63]. If the memory controller issues a *READ* command before t_{RCD} elapses, the bitline voltage may be below V_{read}, which can lead to the reading of a wrong value.

To return a cell to its precharged state, the voltage in the cell must first be fully restored. A cell is expected to be fully restored when the memory controller satisfies a time interval dictated by t_{RAS} after issuing the *ACT* command. Failing to satisfy t_{RAS} may lead to insufficient amount of charge to be restored in the cells of the accessed row. A subsequent activation of the row can then result in the reading of incorrect data from the cells.

Once the cell is successfully *restored* (④), the memory controller can issue a *PRE* command to close the currently-open row to prepare the bank for an access to another row. The cell returns to the precharged state (⑤) after waiting for the timing parameter t_{RP} following the *PRE* command. Violating t_{RP} may prevent the sense amplifiers from fully driving the bitline back to $\frac{V_{dd}}{2}$, which may later result in the row

to be activated with too small amount of charge in its cells, potentially preventing the sense amplifiers to read the data correctly.

For correct DRAM operation, it is critical for the memory controller to ensure that the DRAM timing parameters defined in the DRAM specification are *not* violated. Violation of the timing parameters may lead to incorrect data to be read from the DRAM, and thus cause unexpected program behavior [26, 27, 30, 52, 67, 84, 87]. In this work, we study the failure modes due to violating DRAM timing parameters and explore their application to reliably generating true random numbers.

2.2. True Random Number Generators

A *true random number generator (TRNG)* requires physical processes (e.g., radioactive decay, thermal noise, Poisson noise) to construct a bitstream of random data. Unlike pseudo-random number generators, the random numbers generated by a TRNG do *not* depend on the previously-generated numbers and *only* depend on the random noise obtained from physical processes. TRNGs are usually validated using statistical tests such as NIST [122] or DIEHARD [97]. A TRNG typically consists of 1) an *entropy source*, 2) a *randomness extraction technique*, and sometimes 3) a *post-processor*, which improves the randomness of the extracted data often at the expense of throughput. These three components are typically used to reliably generate true random numbers [135, 139].

Entropy Source. The entropy source is a critical component of a random number generator, as its amount of entropy affects the unpredictability and the throughput of the generated random data. Various physical phenomena can be used as entropy sources. In the domain of electrical circuits, thermal and Poisson noise, jitter, and circuit metastability have been proposed as processes that have high entropy [16, 22, 55, 56, 101, 119, 146, 151, 153]. To ensure robustness, the entropy source should not be visible or modifiable by an adversary. Failing to satisfy that requirement would result in generating predictable data, and thus put the system into a state susceptible to security attacks.

Randomness Extraction Technique. The randomness extraction technique harvests random data from an entropy source. A good randomness extraction technique should have two key properties. First, it should have high throughput, i.e., extract as much as randomness possible in a short amount of time [79, 135], especially important for applications that require high-throughput random number generation (e.g., security applications [13, 15, 21, 31, 37, 47, 69, 80, 82, 95, 101, 121, 135, 142, 152, 159, 162], scientific simulation [21, 95]). Second, it should not disturb the physical process [79, 135]. Affecting the entropy source during the randomness extraction process would make the harvested data predictable, lowering the reliability of the TRNG.

Post-processing. Harvesting randomness from a physical phenomenon *may* produce bits that are biased or correlated [79, 118]. In such a case, a post-processing step, which is also known as *de-biasing*, is applied to eliminate the bias and correlation. The post-processing step also provides protection against environmental changes and adversary tampering [79, 118, 135]. Well-known post-processing techniques are the von Neumann corrector [64] and cryptographic hash functions such as SHA-1 [38] or MD5 [120]. These post-processing steps work well, but generally result in decreased throughput (e.g., up to 80% [81]).

3. Motivation and Goal

True random numbers sampled from physical phenomena have a number of real-world applications from system security [13, 121, 135] to recreational entertainment [135]. As user data privacy becomes a *highly-sought* commodity in Internet-of-Things (IoT) and mobile devices, enabling primitives that provide security on such systems becomes critically important [90, 115, 162]. Cryptography is one typical method for securing systems against various attacks by encrypting the system's data with keys generated with true random values. Many cryptographic algorithms require random values to generate keys in many standard protocols (e.g., TLS/SSL/RSA/VPN keys) to either 1) encrypt network packets, file systems, and data, 2) select internet protocol sequence numbers (TCP), or 3) generate data padding values [31, 37, 47, 69, 80, 82, 152, 162]. TRNGs are also commonly used in authentication protocols and in countermeasures against hardware attacks [31], in which psuedo-random number generators (PRNGs) are shown to be insecure [31, 152]. To keep up with the *ever-increasing* rate of secure data creation, especially with the growing number of commodity data-harvesting devices (e.g., IoT and mobile devices), the ability to generate true random numbers with *high throughput and low latency* becomes ever more relevant to maintain user data privacy. In addition, *high-throughput* TRNGs are already *essential* components of various important applications such as scientific simulation [21, 95], industrial testing, statistical sampling, randomized algorithms, and recreational entertainment [13, 15, 21, 95, 101, 121, 135, 142, 159, 162].

A *widely-available, high-throughput, low-latency* TRNG will enable all previously mentioned applications that rely on TRNGs, including improved security and privacy in most systems that are known to be vulnerable to attacks [90, 115, 162], as well as enable research that we may not anticipate at the moment. One such direction is using a one-time pad (i.e., a private key used to encode and decode only a single message) with quantum key distribution, which requires at least *4Gb/s* of true random number generation throughput [34, 94, 154]. Many *high-throughput* TRNGs have been recently proposed [12, 15, 31, 42, 48, 80, 82, 101, 110, 147, 154, 159, 162, 163], and the availability of these high-throughput TRNGs can enable a wide range of new applications with improved security and privacy.

DRAM offers a promising substrate for developing an effective and widely-available TRNG due to the prevalence of DRAM throughout all modern computing systems ranging from microcontrollers to supercomputers. A high-throughput DRAM-based TRNG would help enable widespread adoption of applications that are today limited to only select architectures equipped with dedicated high-performance TRNG engines. Examples of such applications include high-performance scientific simulations and cryptographic applications for securing devices and communication protocols, both of which would run much more efficiently on mobile devices, embedded devices, or microcontrollers with the availability of higher-throughput TRNGs in the system.

In terms of the CPU architecture itself, a high-throughput DRAM-based TRNG could help the memory controller to improve scheduling decisions [10, 74, 107, 108, 136, 137, 138, 150] and enable the implementation a truly-randomized version of PARA [73] (i.e., a protection mechanism against the RowHammer vulnerability [73, 106]). Furthermore, a DRAM-based TRNG would likely have additional hardware and software applications as system designs become more capable and increasingly security-critical.

In addition to traditional computing paradigms, DRAM-based TRNGs can benefit processing-in-memory (PIM) architectures [45, 130], which co-locate logic within or near memory to overcome the large bandwidth and energy bottleneck caused by the memory bus and leverage the *significant* data parallelism available within the DRAM chip itself. Many prior works provide primitives for PIM or exploit PIM-enabled sys-

tems for workload acceleration [4, 5, 11, 19, 20, 29, 41, 43, 44, 45, 53, 58, 59, 70, 86, 93, 103, 113, 125, 126, 127, 128, 129, 130, 140, 160]. A low-latency, high-throughput DRAM-based TRNG can enable PIM applications to source random values *directly within the memory itself*, thereby enhancing the overall potential, security, and privacy, of PIM-enabled architectures. For example, in applications that require true random numbers, a DRAM-based TRNG can enable large contiguous code segments to execute in memory, which would reduce communication with the CPU, and thus improve system efficiency. A DRAM-based TRNG can also enable security tasks to run completely in memory. This would remove the dependence of PIM-based security tasks on an I/O channel and would increase overall system security.

We posit, based on analysis done in prior works [64,79,124], that an *effective* TRNG must satisfy *six* key properties: it must 1) have low implementation cost, 2) be fully non-deterministic such that it is impossible to predict the next output given complete information about how the mechanism operates, 3) provide a continuous stream of true random numbers with high throughput, 4) provide true random numbers with low latency, 5) exhibit low system interference, i.e., not significantly slow down concurrently-running applications, and 6) generate random values with low energy overhead.

To this end, our **goal** in this work, is to provide a widely-available TRNG for DRAM devices that satisfies all six key properties of an effective TRNG.

4. Testing Environment

In order to test our hypothesis that DRAM cells are an effective source of entropy when accessed with reduced DRAM timing parameters, we developed an infrastructure to characterize modern LPDDR4 DRAM chips. We also use an infrastructure for DDR3 DRAM chips, SoftMC [52, 132], to demonstrate empirically that our proposal is applicable beyond the LPDDR4 technology. Both testing environments give us precise control over DRAM commands and DRAM timing parameters as verified with a logic analyzer probing the command bus.

We perform all tests, unless otherwise specified, using a total of 282 2y-nm LPDDR4 DRAM chips from three major manufacturers in a thermally-controlled chamber held at 45°C. For consistency across results, we precisely stabilize the ambient temperature using heaters and fans controlled via a microcontroller-based proportional-integral-derivative (PID) loop to within an accuracy of 0.25°C and a reliable range of 40°C to 55°C. We maintain DRAM temperature at 15°C above ambient temperature using a separate local heating source. We use temperature sensors to smooth out temperature variations caused by self-induced heating.

We also use a separate infrastructure, based on open-source SoftMC [52, 132], to validate our mechanism on 4 DDR3 DRAM chips from a single manufacturer. SoftMC enables precise control over timing parameters, and we house the DRAM chips inside another temperature chamber to maintain a stable ambient testing temperature (with the same temperature range as the temperature chamber used for the LPDDR4 devices).

To explore the various effects of temperature, short-term aging, and circuit-level interference (in Section 5) on activation failures, we reduce the t_{RCD} parameter from the default $18ns$ to $10ns$ for all experiments, unless otherwise stated. Algorithm 1 explains the general testing methodology we use to induce activation failures. First, we write a data pattern to the region of DRAM under test (Line 2). Next, we reduce the t_{RCD} parameter to begin inducing activation failures (Line 3). We then access the DRAM region in column order (Lines 4-5) in order to ensure that each DRAM access is to a closed

Algorithm 1: DRAM Activation Failure Testing

```
1  DRAM_ACT_failure_testing(data_pattern, DRAM_region):
2    write data_pattern (e.g., solid 1s) into all cells in DRAM_region
3    set low tRCD for ranks containing DRAM_region
4    foreach col in DRAM_region:
5      foreach row in DRAM_region:
6        activate(row)    // fully refresh cells
7        precharge(row)   // ensure next access activates the row
8        activate(row)
9        read(col)        // induce activation failure on col
10       precharge(row)
11       record activation failures to storage
12   set default tRCD for DRAM ranks containing DRAM_region
```

DRAM row and thus requires an activation. This enables each access to induce activation failures in DRAM. Prior to each reduced-latency read, we first refresh the target row such that each cell has the same amount of charge each time it is accessed with a reduced-latency read. We effectively refresh a row by issuing an activate (Line 6) followed by a precharge (Line 7) to that row. We then induce the activation failures by issuing consecutive activate (Line 8), read (Line 9), and precharge (Line 10) commands. Afterwards, we record any activation failures that we observe (Line 11). We find that this methodology enables us to quickly induce activation failures across *all* of DRAM, and minimizes testing time.

5. Activation Failure Characterization

To demonstrate the viability of using DRAM cells as an entropy source for random data, we explore and characterize DRAM failures when employing a reduced DRAM activation latency (t_{RCD}) across 282 LPDDR4 DRAM chips. We also compare our findings against those of prior works that study an older generation of DDR3 DRAM chips [27, 71, 84, 87] to cross-validate our infrastructure. To understand the effects of changing environmental conditions on a DRAM cell that is used as a source of entropy, we rigorously characterize DRAM cell behavior as we vary four environmental conditions. First, we study the effects of DRAM array design-induced variation (i.e., the spatial distribution of activation failures in DRAM). Second, we study data pattern dependence (DPD) effects on DRAM cells. Third, we study the effects of temperature variation on DRAM cells. Fourth, we study a DRAM cell's activation failure probability over time. We present several key observations that support the viability of a mechanism that generates random numbers by accessing DRAM cells with a reduced t_{RCD}. In Section 6, we discuss a mechanism to effectively sample DRAM cells to extract true random numbers while minimizing the effects of environmental condition variation (presented in this section) on the DRAM cells.

5.1. Spatial Distribution of Activation Failures

To study which regions of DRAM are better suited to generating random data, we first visually inspect the spatial distributions of activation failures both across DRAM chips and within each chip individually. Figure 4 plots the spatial distribution of activation failures in a *representative* 1024×1024 array of DRAM cells taken from a single DRAM chip. Every observed activation failure is marked in black. We make two observations. First, we observe that each contiguous region of 512 DRAM rows[2] consists of repeating rows with the same set (or subset) of column bits that are prone to activation failures. As shown in the figure, rows 0 to 511 have the same 8 (or a subset of the 8) column bits failing in the row, and rows 512 to 1023 have the same 4 (or a subset of the 4) column

[2]We note that subarrays have either 512 or 1024 (not shown) rows depending on the manufacturer of the DRAM device.

bits failing in the row. We hypothesize that these contiguous regions reveal the DRAM subarray architecture as a result of variation across the local sense amplifiers in the subarray. We indicate the two subarrays in Figure 4 as Subarray A and Subarray B. A "weaker" local sense amplifier results in cells that share its respective *local bitline* in the subarray having an increased probability of failure. For this reason, we observe that activation failures are localized to a few columns within a DRAM subarray as shown in Figure 4. Second, we observe that within a subarray, the activation failure probability increases across rows (i.e., activation failures are *more* likely to occur in higher-numbered rows in the subarray and are *less* likely in lower-numbered rows in the subarray). This can be seen from the fact that more cells fail in higher-numbered rows in the subarray (i.e., there are more black marks higher in each subarray). We hypothesize that the failure probability of a cell attached to a local bitline correlates with the distance between the row and the local sense amplifiers, and further rows have less time to amplify their data due to the signal propagation delay in a bitline. These observations are similar to those made in prior studies [27, 71, 84, 87] on DDR3 devices.

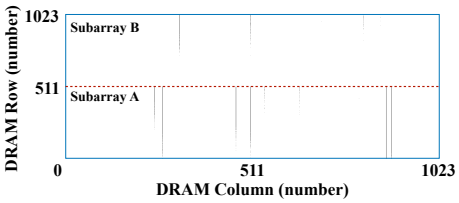


Figure 4: Activation failure bitmap in 1024×1024 cell array.

We next study the granularity at which we can induce activation failures when accessing a row. We observe (not shown) that activation failures occur *only* within the first cache line that is accessed immediately following an activation. No subsequent access to an already *open* row results in activation failures. This is because cells within the *same* row have a longer time to restore their cell charge (Figure 3) when they are accessed after the row has already been opened. We draw two key conclusions: 1) the region *and* bitline of DRAM being accessed affect the number of observable activation failures, and 2) different DRAM subarrays *and* different local bitlines exhibit varying levels of entropy.

5.2. Data Pattern Dependence

To understand the data pattern dependence of activation failures and DRAM cell entropy, we study how effectively we can discover failures using different data patterns across multiple rounds of testing. Our *goal* in this experiment is to determine which data pattern results in the highest entropy such that we can generate random values with high throughput. Similar to prior works [91, 112] that extensively describe the data patterns, we analyze a total of 40 unique data patterns: solid 1s, checkered, row stripe, column stripe, 16 walking 1s, *and* the inverses of all 20 aforementioned data patterns.

Figure 5 plots the ratio of activation failures discovered by a particular data pattern after 100 iterations of Algorithm 1 relative to the *total* number of failures discovered by *all* patterns for a representative chip from each manufacturer. We call this metric *coverage* because it indicates the effectiveness of a single data pattern to identify all possible DRAM cells that are prone to activation failure. We show results for each pattern individually except for the WALK1 and WALK0 patterns, for which we show the mean (bar) and minimum/maximum (error bars) coverage across all 16 iterations of each walking pattern.

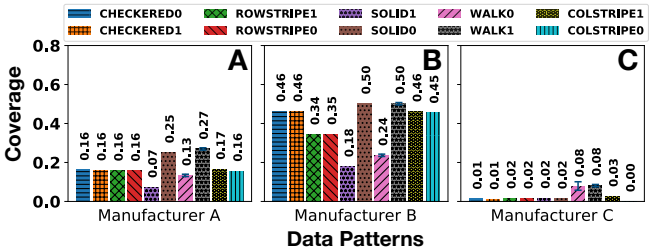


Figure 5: Data pattern dependence of DRAM cells prone to activation failure over 100 iterations

We make three key observations from this experiment. First, we find that testing with different data patterns identifies different subsets of the total set of possible activation failures. This indicates that 1) different data patterns cause different DRAM cells to fail and 2) specific data patterns induce more activation failures than others. Thus, certain data patterns may extract more entropy from a DRAM cell array than other data patterns. Second, we find that, of *all* 40 tested data patterns, each of the 16 *walking 1s*, for a given device, provides a similarly high coverage, regardless of the manufacturer. This high coverage is similarly provided by only one other data pattern per manufacturer: solid 0s for manufacturers A and B, and walking 0s for manufacturer C. Third, if we repeat this experiment (i.e., Figure 5) while varying the number of iterations of Algorithm 1, the *total failure count* across all data patterns *increases* as we increase the number of iterations of Algorithm 1. This indicates that not all DRAM cells fail deterministically when accessed with a reduced t_{RCD}, providing a potential source of entropy for random number generation.

We next analyze each cell's probability of failing when accessed with a reduced t_{RCD} (i.e., its *activation failure probability*) to determine which data pattern most effectively identifies cells that provide high entropy. We note that DRAM cells with an activation failure probability F_{prob} of 50% provide high entropy when accessed many times. With the same data used to produce Figure 5, we study the different data patterns with regard to the number of cells they cause to fail 50% of the time. Interestingly, we find that the data pattern that induces the most failures overall does not necessarily find the most number of cells that fail 50% of the time. In fact, when searching for cells with an F_{prob} between 40% and 60%, we observe that the data patterns that find the highest number of cells are solid 0s, checkered 0s, and solid 0s for manufacturers A, B, and C, respectively. We conclude that: 1) due to manufacturing and design variation across DRAM devices from different manufacturers, different data patterns result in different failure probabilities in our DRAM devices, and 2) to provide high entropy when accessing DRAM cells with a reduced t_{RCD}, we should use the respective data pattern that finds the most number of cells with an F_{prob} of 50% for DRAM devices from a given manufacturer.

Unless otherwise stated, in the rest of this paper, we use the solid 0s, checkered 0s, and solid 0s data patterns for manufacturers A, B, and C, respectively, to analyze F_{prob} at the granularity of a single cell and to study the effects of temperature and time on our sources of entropy.

5.3. Temperature Effects

In this section, we study whether temperature fluctuations affect a DRAM cell's activation failure probability and thus the entropy that can be extracted from the DRAM cell. To analyze temperature effects, we record the F_{prob} of cells throughout our DRAM devices across 100 iterations of Algorithm 1 at

5°C increments between 55°C and 70°). Figure 6 aggregates results across 30 DRAM modules from each DRAM manufacturer. Each point in the figure represents how the F_{prob} of a DRAM cell changes as the temperature changes (i.e., ΔF_{prob}). The x-axis shows the F_{prob} of a single cell at temperature T (i.e., the baseline temperature), and the y-axis shows the F_{prob} of the same cell at temperature $T + 5$ (i.e., 5°C above the baseline temperature). Because we test each cell at each temperature across 100 iterations, the granularity of F_{prob} on both the x- and y-axes is 1%. For a given F_{prob} at temperature T (x% on the x-axis), we aggregate *all* respective F_{prob} points at temperature $T+5$ (y% on the y-axis) with box-and-whiskers plots[3] to show how the given F_{prob} is affected by the increased DRAM temperature. The *box* is drawn in blue and contains the *median* drawn in red. The *whiskers* are drawn in gray, and the *outliers* are indicated with orange pluses.

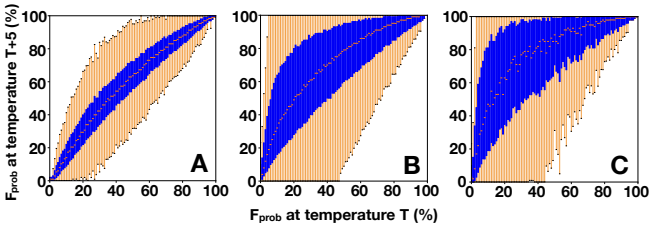


Figure 6: Effect of temperature variation on failure probability

We observe that F_{prob} at temperature $T + 5$ tends to be higher than F_{prob} at temperature T, as shown by the blue region of the figure (i.e., the boxes of the box-and-whiskers plots) lying above the $x = y$ line. However, fewer than 25% of all data points fall below the $x = y$ line, indicating that a portion of cells have a lower F_{prob} as temperature is increased.

We observe that DRAM devices from different manufacturers are affected by temperature differently. DRAM cells of manufacturer A have the *least* variation of ΔF_{prob} when temperature is increased since the boxes of the box-and-whiskers plots are strongly correlated with the $x = y$ line. How a DRAM cell's activation failure probability changes in DRAM devices from *other* manufacturers is unfortunately *less* predictable under temperature change (i.e., a DRAM cell from manufacturers B or C has higher variation in F_{prob} change), but the data still shows a strong positive correlation between temperature and F_{prob}. We conclude that temperature affects cell failure probability (F_{prob}) to different degrees depending on the manufacturer of the DRAM device, but increasing temperature generally increases the activation failure probability.

5.4. Entropy Variation over Time

To determine whether the failure probability of a DRAM cell changes over time, we complete 250 *rounds* of recording the activation failure probability of DRAM cells over the span of 15 days. Each round consists of accessing every cell in DRAM 100 times with a reduced t_{RCD} value and recording the failure probability for each individual cell (out of 100 iterations). We find that a DRAM cell's activation failure probability does *not* change significantly over time. This means that, once we identify a DRAM cell that exhibits high entropy, we can rely on the cell to maintain its high entropy over time. We hypothesize that this is because a DRAM cell fails with high entropy when process manufacturing variation in peripheral and DRAM cell circuit elements combine such that, when we read the cell using a reduced t_{RCD} value, we induce a metastable state resulting from the cell voltage falling between the reliable sensing margins (i.e., falling close to $\frac{V_{dd}}{2}$) [27]. Since manufacturing variation is fully determined at manufacturing time, a DRAM cell's activation failure probability is stable over time given the same experimental conditions. In Section 6.1, we discuss our methodology for selecting DRAM cells for extracting stable entropy, such that we can preemptively avoid longer-term aging effects that we do not study in this paper.

[3] A box-and-whiskers plot emphasizes the important metrics of a dataset's distribution. The box is lower-bounded by the first quartile (i.e., the median of the first half of the ordered set of data points) and upper-bounded by the third quartile (i.e., the median of the second half of the ordered set of data points). The median falls within the box. The *inter-quartile range* (IQR) is the distance between the first and third quartiles (i.e., box size). Whiskers extend an additional $1.5 \times IQR$ on either sides of the box. We indicate outliers, or data points outside of the range of the whiskers, with pluses.

6. D-RaNGe: A DRAM-based TRNG

Based on our rigorous analysis of DRAM activation failures (presented in Section 5), we propose D-RaNGe, a flexible mechanism that provides high-throughput DRAM-based true random number generation (TRNG) by sourcing entropy from a subset of DRAM cells and is built fully within the memory controller. D-RaNGe is based on the **key observation** that DRAM cells fail probabilistically when accessed with reduced DRAM timing parameters, and this probabilistic failure mechanism can be used as a source of true random numbers. While there are many other timing parameters that we could reduce to induce failures in DRAM [26, 27, 71, 84, 85, 87], we focus specifically on reducing t_{RCD} below manufacturer-recommended values to study the resulting activation failures.[4]

Activation failures occur as a result of reading the value from a DRAM cell *too soon* after sense amplification. This results in reading the value at the sense amplifiers before the bitline voltage is amplified to an I/O-readable voltage level. The probability of reading incorrect data from the DRAM cell therefore depends largely on the bitline's voltage at the time of reading the sense amplifiers. Because there is significant process variation across the DRAM cells and I/O circuitry [27, 71, 84, 87], we observe a wide variety of failure probabilities for different DRAM cells (as discussed in Section 5) for a given t_{RCD} value, ranging from 0% probability to 100% probability.

We discover that a subset of cells fail at ∼50% probability, and a subset of these cells fail randomly with high entropy (shown in Section 7.2). In this section, we first discuss our method of identifying such cells, which we refer to as *RNG cells* (in Section 6.1). Second, we describe the mechanism with which D-RaNGe *samples* RNG cells to extract random data (Section 6.2). Finally, we discuss a potential design for integrating D-RaNGe in a full system (Section 6.3).

6.1. RNG Cell Identification

Prior to generating random data, we must first identify cells that are capable of producing truly random output (i.e., RNG cells). Our process of identifying RNG cells involves reading every cell in the DRAM array 1000 times with a *reduced* t_{RCD} and approximating each cell's Shannon entropy [131] by counting the occurrences of 3-bit symbols across its 1000-bit stream. We identify cells that generate an approximately equal number of every possible 3-bit symbol (±10% of the number of expected symbols) as RNG cells.

[4] We believe that reducing other timing parameters could be used to generate true random values, but we leave their exploration to future work.

We find that RNG cells provide unbiased output, meaning that a post-processing step (described in Section 2.2) is *not* necessary to provide sufficiently high entropy for random number generation. We also find that RNG cells *maintain high entropy across system reboots*. In order to account for our observation that entropy from an RNG cell changes depending on the DRAM temperature (Section 5.3), we identify reliable RNG cells at each temperature and store their locations in the memory controller. Depending on the DRAM temperature at the time an application requests random values, D-RaNGe samples the appropriate RNG cells. To ensure that DRAM aging does not negatively impact the reliability of RNG cells, we require re-identifying the set of RNG cells at regular intervals. From our observation that entropy does not change significantly over a tested 15 day period of sampling RNG cells (Section 5.4), we expect the interval of re-identifying RNG cells to be at least 15 days long. Our RNG cell identification process is effective at identifying cells that are reliable entropy sources for random number generation, and we quantify their randomness using the NIST test suite for randomness [122] in Section 7.1.

6.2. Sampling RNG Cells for Random Data

Given the availability of these RNG cells, we use our observations in Section 5 to design a high-throughput TRNG that quickly and repeatedly samples RNG cells with reduced DRAM timing parameters. Algorithm 2 demonstrates the key components of D-RaNGe that enable us to generate random numbers with high throughput. D-RaNGe takes in

Algorithm 2: D-RaNGe: A DRAM-based TRNG

```
1  D-RaNGe(num_bits):   // num_bits: number of random bits requested
2    DP: a known data pattern that results in high entropy
3    select 2 DRAM words with RNG cells in distinct rows in each bank
4    write DP to chosen DRAM words and their neighboring cells
5    get exclusive access to rows of chosen DRAM words and nearby cells
6    set low t_RCD for DRAM ranks containing chosen DRAM words
7    for each bank:
8      read data in DRAM word 1 (DW_1)  // induce activation failure
9      write the read value of DW_1's RNG cells to bitstream
10     write original data value back into DW_1
11     memory barrier          // ensure completion of write to DW_1
12     read data in DRAM word 2 (DW_2)  // induce activation failure
13     write the read value of DW_2's RNG cells to bitstream
14     write original data value back into DW_2
15     memory barrier          // ensure completion of write to DW_2
16     if bitstream_size ≥ num_bits:
17       break
18   set default t_RCD for DRAM ranks of the chosen DRAM words
19   release exclusive access to rows of chosen words and nearby cells
```

num_bits as an argument, which is defined as the number of random bits desired (Line 1). D-RaNGe then prepares to generate random numbers in Lines 2-6 by first selecting DRAM words (i.e., the granularity at which a DRAM module is accessed) containing known RNG cells for generating random data (Line 3). To maximize the throughput of random number generation, D-RaNGe chooses DRAM words with the highest density of RNG cells in each bank (to exploit DRAM parallelism). Since each DRAM access can induce activation failures *only* in the accessed DRAM word, the density of RNG cells per DRAM word determines the number of random bits D-RaNGe can generate per access. For each available DRAM bank, D-RaNGe selects two DRAM words (in distinct DRAM rows) containing RNG cells. The purpose of selecting two DRAM words in *different* rows is to *repeatedly* cause *bank conflicts*, or issue requests to *closed* DRAM rows so that every read request will *immediately* follow an activation. This is done by alternating accesses to the chosen DRAM words in different DRAM rows. After selecting DRAM words for generating random values, D-RaNGe writes a known data pattern that results in high entropy to each chosen DRAM word and its neighboring cells (Line 4) and gains exclusive access to rows containing the two chosen DRAM words as well as their neighboring cells (Line 5).[5] This ensures that the data pattern surrounding the RNG cell and the original value of the RNG cell stay constant prior to each access such that the failure probability of each RNG cell remains reliable (as observed to be necessary in Section 5.2). To begin generating random data (i.e., sampling RNG cells), D-RaNGe reduces the value of t_{RCD} (Line 6). From every available bank (Line 7), D-RaNGe generates random values in parallel (Lines 8-15). Lines 8 and 12 indicate the commands to alternate accesses to two DRAM words in distinct rows of a bank to both 1) induce activation failures and 2) precharge the recently-accessed row. After inducing activation failures in a DRAM word, D-RaNGe extracts the value of the RNG cells within the DRAM word (Lines 9 and 13) to use as random data and restores the DRAM word to its original data value (Lines 10 and 14) to maintain the original data pattern. Line 15 ensures that writing the original data value is complete before attempting to sample the DRAM words again. Lines 16 and 17 simply end the loop if enough random bits of data have been harvested. Line 18 sets the t_{RCD} timing parameter back to its default value, so other applications can access DRAM without corrupting data. Line 19 releases exclusive access to the rows containing the chosen DRAM words and their neighboring rows.

We find that this methodology maximizes the opportunity for activation failures in DRAM, thereby maximizing the rate of generating random data from RNG cells.

6.3. Full System Integration

In this work, we focus on developing a flexible substrate for sampling RNG cells fully from within the memory controller. D-RaNGe generates random numbers using a simple firmware routine running entirely within the memory controller. The firmware executes the sampling algorithm (Algorithm 2) whenever an application requests random samples and there is available DRAM bandwidth (i.e., DRAM is not servicing other requests or maintenance commands). In order to minimize latency between requests for samples and their corresponding responses, a small queue of already-harvested random data may be maintained in the memory controller for use by the system. Overall performance overhead can be minimized by tuning both 1) the queue size and 2) how the memory controller prioritizes requests for random numbers relative to normal memory requests.

In order to integrate D-RaNGe with the rest of the system, the system designer needs to decide how to best expose an interface by which an application can leverage D-RaNGe to generate true random numbers on their system. There are many ways to achieve this, including, but not limited to:

- Providing a simple `REQUEST` and `RECEIVE` interface for applications to request and receive the random numbers using memory-mapped configuration status registers (CSRs) [156] or other existing I/O datapaths (e.g., x86 `IN` and `OUT` opcodes, Local Advanced Programmable Interrupt Controller (LAPIC configuration [61]).

[5] Ensuring exclusive access to DRAM rows can be done by remapping rows to 1) redundant DRAM rows or 2) buffers in the memory controller so that these rows are hidden from the system software and only accessible by the memory controller for generating random numbers.

- Adding a new ISA instruction (e.g., Intel `RDRAND` [49]) that retrieves random numbers from the memory controller and stores them into processor registers.

The operating system may then expose one or more of these interfaces to user applications through standard kernel-user interfaces (e.g., system calls, file I/O, operating system APIs). The system designer has complete freedom to choose between these (and other) mechanisms that expose an interface for user applications to interact with D-RaNGe. We expect that the best option will be system specific, depending both on the desired D-RaNGe use cases and the ease with which the design can be implemented.

7. D-RaNGe Evaluation

We evaluate three key aspects of D-RaNGe. First, we show that the random data obtained from RNG cells identified by D-RaNGe passes all of the tests in the NIST test suite for randomness (Section 7.1). Second, we analyze the existence of RNG cells across 59 LPDDR4 and 4 DDR3 DRAM chips (due to long testing time) randomly sampled from the overall population of DRAM chips across all three major DRAM manufacturers (Section 7.2). Third, we evaluate D-RaNGe in terms of the six key properties of an ideal TRNG as explained in Section 3 (Section 7.3).

7.1. NIST Tests

First, we identify RNG cells using our RNG cell identification process (Section 6.1). Second, we sample *each* identified RNG cell one million times to generate large amounts of random data (i.e., 1 Mb *bitstreams*). Third, we evaluate the entropy of the bitstreams from the identified RNG cells with the NIST test suite for randomness [122]. Table 1 shows the average results of 236 1 Mb bitstreams[6] across the 15 tests of the full NIST test suite for randomness. P-values are calculated

NIST Test Name	P-value	Status
monobit	0.675	PASS
frequency_within_block	0.096	PASS
runs	0.501	PASS
longest_run_ones_in_a_block	0.256	PASS
binary_matrix_rank	0.914	PASS
dft	0.424	PASS
non_overlapping_template_matching	>0.999	PASS
overlapping_template_matching	0.624	PASS
maurers_universal	0.999	PASS
linear_complexity	0.663	PASS
serial	0.405	PASS
approximate_entropy	0.735	PASS
cumulative_sums	0.588	PASS
random_excursion	0.200	PASS
random_excursion_variant	0.066	PASS

Table 1: D-RaNGe results with NIST randomness test suite.

for each test,[7] where the null hypothesis for each test is that a perfect random number generator would *not* have produced random data with *better* characteristics for the given test than the tested sequence [99]. Since the resulting P-values for each test in the suite are greater than our chosen level of significance, $\alpha = 0.0001$, we accept our null hypothesis for each test. We note that all 236 bitstreams pass all 15 tests with similar P-values. Given our $\alpha = 0.0001$, our proportion of passing sequences (1.0) falls within the range of acceptable proportions of sequences that pass each test ([0.998,1] calculated by the NIST statistical test suite using $(1-\alpha) \pm 3\sqrt{\frac{\alpha(1-\alpha)}{k}}$, where k is the number of tested sequences). This *strongly* indicates that D-RaNGe can generate unpredictable, truly random values. Using the proportion of 1s and 0s generated from each RNG cell, we calculate Shannon entropy [131] and find the *minimum* entropy across all RNG cells to be 0.9507.

[6]We test data obtained from 4 RNG cells from each of 59 DRAM chips, to maintain a reasonable NIST testing time and thus show that RNG cells across all tested DRAM chips reliably generate random values.

[7]A p-value close to 1 indicates that we must accept the null hypothesis, while a p-value close to 0 and below a small threshold, e.g., $\alpha = 0.0001$ (recommended by the NIST Statistical Test Suite documentation [122]), indicates that we must reject the null hypothesis.

7.2. RNG Cell Distribution

The throughput at which D-RaNGe generates random numbers is a function of the 1) density of RNG cells per DRAM word and 2) bandwidth with which we can access DRAM words when using our methodology for inducing activation failures. Since each DRAM access can induce activation failures *only* in the accessed DRAM word, the density of RNG cells per DRAM word indicates the number of random bits D-RaNGe can sample per access. We first study the density of RNG cells per word across DRAM chips. Figure 7 plots the distribution of the number of words containing x RNG cells (indicated by the value on the x-axis) per *bank* across 472 banks from 59 DRAM devices from all manufacturers. The distribution is presented as a box-and-whiskers plot where the y-axis has a logarithmic scale with a 0 point. The three plots respectively show the distributions for DRAM devices from the three manufacturers (indicated at the bottom left corner of each plot).

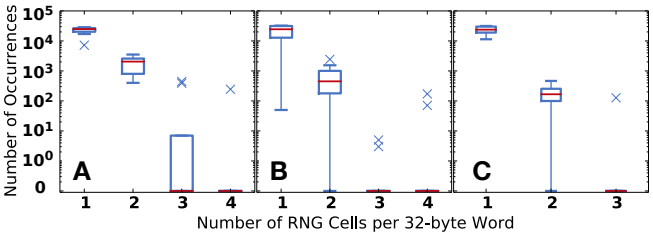


Figure 7: Density of RNG cells in DRAM words per bank.

We make three key observations. First, RNG cells are *widely available in every bank* across many chips. This means that we can use the available DRAM access parallelism that multiple banks offer and sample RNG cells from each DRAM bank in parallel to improve random number generation throughput. Second, *every* bank that we analyze has *multiple DRAM words* containing at least one RNG cell. The DRAM bank with the smallest occurrence of RNG cells has 100 DRAM words containing only 1 RNG cell (manufacturer B). Discounting this point, the distribution of the number of DRAM words containing only 1 RNG cell is tight with a high number of RNG cells (e.g., tens of thousands) in each bank, regardless of the manufacturer. Given our random sample of DRAM chips, we expect that the existence of RNG cells in DRAM banks will hold true for all DRAM chips. Third, we observe that a single DRAM word can contain as many as 4 RNG cells. Because the throughput of accesses to DRAM is fixed, the number of RNG cells in the accessed words essentially acts as a multiplier for the throughput of random numbers generated (e.g., accessing DRAM words containing 4 RNG cells results in *4x* the throughput of random numbers compared to accessing DRAM words containing 1 RNG cell).

7.3. TRNG Key Characteristics Evaluation

We now evaluate D-RaNGe in terms of the six key properties of an effective TRNG as explained in Section 3.

Low Implementation Cost. To induce activation failures, we must be able to reduce the DRAM timing parameters below manufacturer-specified values. Because memory controllers issue memory accesses according to the timing

parameters specified in a set of internal registers, D-RaNGe requires simple software support to be able to programmatically modify the memory controller's registers. Fortunately, there exist some processors [7, 8, 40, 84] that *already* enable software to directly change memory controller register values, i.e., the DRAM timing parameters. These processors can easily generate random numbers with D-RaNGe.

All other processors that do *not* currently support direct changes to memory controller registers require *minimal* software changes to expose an interface for changing the memory controller registers [9, 52, 123, 132]. To enable a more efficient implementation, the memory controller could be programmed such that it issues DRAM accesses with distinct timing parameters on a per-access granularity to reduce the overhead in 1) changing the DRAM timing parameters and 2) allow concurrent DRAM accesses by other applications. In the rare case where these registers are unmodifiable by even the hardware, the hardware changes necessary to enable register modification are minimal and are simple to implement [52, 84, 132].

We experimentally find that we can induce activation failures with t_{RCD} between $6ns$ and $13ns$ (reduced from the default of $18ns$). Given this wide range of failure-inducing t_{RCD} values, most memory controllers should be able to adjust their timing parameter registers to a value within this range.

Fully Non-deterministic. As we have shown in Section 7.1, the bitstreams extracted from the D-RaNGe-identified RNG cells pass *all* 15 NIST tests. We have full reason to believe that we are inducing a metastable state of the sense amplifiers (as hypothesized by [27]) such that we are effectively sampling random physical phenomena to extract unpredictable random values.

High Throughput of Random Data. Due to the various use cases of random number generation discussed in Section 3, different applications have different throughput requirements for random number generation, and applications may tolerate a reduction in performance so that D-RaNGe can quickly generate true random numbers. Fortunately, D-RaNGe provides flexibility to tradeoff between the *system interference* it causes, i.e., the slowdown experienced by concurrently running applications, and the random number generation throughput it provides. To demonstrate this flexibility, Figure 8 plots the TRNG throughput of D-RaNGe when using varying numbers of banks (x banks on the x-axis) across the three DRAM manufacturers (indicated at the top left corner of each plot). For each number of banks used, we plot the distribution of TRNG throughput that we observe *real* DRAM devices to provide. The available density of RNG cells in a DRAM device (provided in Figure 7) dictates the TRNG throughput that the DRAM device can provide. We plot each distribution as a box-and-whiskers plot. For each number of banks used, we select x banks with the greatest sum of RNG cells across each banks' two DRAM words with the highest density of RNG cells (that are *not* in the same DRAM row). We select two DRAM words per bank because we must alternate accesses between two DRAM rows (as shown in Lines 8 and 12 of Algorithm 2). The sum of the RNG cells available across the two selected DRAM words for each bank is considered each bank's *TRNG data rate*, and we use this value to obtain D-RaNGe's throughput. We use Ramulator [2, 76] to obtain the rate at which we can execute the core loop of Algorithm 2 with varying numbers of banks. We obtain the random number generation throughput for x banks with the following equation:

$$TRNG\_Throughput_{x\_Banks} = \sum_{n=1}^{x} \frac{TRNG\_data\_rate_{Bank\_n}}{Alg2\_Runtime_{x\_banks}} \quad (1)$$

where $TRNG\_data\_rate_{Bank\_n}$ is the TRNG data rate for the selected bank, and $Alg2\_Runtime_{x\_banks}$ is the runtime of the core loop of Algorithm 2 when using x Banks. We note that because we observe small variation in the density of RNG cells per word (between 0 and 4), we see that TRNG throughput across different chips is generally very similar. For this reason, we see that the box and whiskers are condensed into a single point for distributions of manufacturers B and C. We find that when *fully* using *all* 8 banks in a single DRAM channel, every device can produce *at least* 40 Mb/s of random data regardless of manufacturer. The highest throughput we observe from devices of manufacturers A/B/C respectively are 179.4/134.5/179.4 Mb/s. On average, across all manufacturers, we find that D-RaNGe can provide a throughput of 108.9 Mb/s.

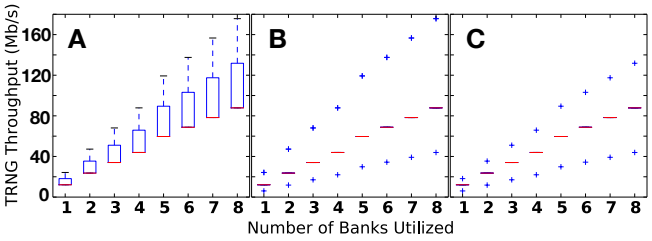


Figure 8: Distribution of TRNG throughput across chips.

We draw two key conclusions. First, due to the *parallelism* of multiple banks, the throughput of random number generation increases linearly as we use more banks. Second, there is variation of TRNG throughput across different DRAM devices, but the medians across manufacturers are very similar.

We note that any throughput sample point on this figure can be multiplied by the number of available channels in a memory hierarchy for a better TRNG throughput estimate for a system with multiple DRAM channels. For an example memory hierarchy comprised of 4 DRAM channels, D-RaNGe results in a maximum (average) throughput of 717.4 Mb/s (435.7 Mb/s).

Low Latency. Since D-RaNGe's sampling mechanism consists of a single DRAM access, the latency of generating random values is directly related to the DRAM access latency. Using the timing parameters specified in the JEDEC LPDDR4 specification [63], we calculate D-RaNGe's latency to generate a 64-bit random value. To calculate the *maximum latency* for D-RaNGe, we assume that 1) each DRAM access provides only 1 bit of random data (i.e., each DRAM word contains *only* 1 RNG cell) and 2) we can use only a single bank within a single channel to generate random data. We find that D-RaNGe can generate 64 bits of random data with a *maximum* latency of $960ns$. If D-RaNGe takes full advantage of DRAM's channel- and bank-level parallelism in a system with 4 DRAM channels and 8 banks per channel, D-RaNGe can generate 64 bits of random data by issuing 16 DRAM accesses per channel in parallel. This results in a latency of $220ns$. To calculate the *empirical minimum latency* for D-RaNGe, we fully parallelize D-RaNGe across banks in all 4 channels while also assuming that each DRAM access provides 4 bits of random data, since we find a maximum density of 4 RNG cells per DRAM word in the LPDDR4 DRAM devices that we characterize (Figure 7). We find the empirical minimum latency to be *only* $100ns$ in our tested devices.

Low System Interference. The flexibility of using a different number of banks across the available channels in a system's memory hierarchy allows D-RaNGe to cause varying levels of system interference at the expense of TRNG throughput. This enables application developers to generate random values with D-RaNGe at varying tradeoff points

depending on the running applications' memory access requirements. We analyze D-RaNGe's system interference with respect to DRAM storage overhead and DRAM latency.

In terms of storage overhead, D-RaNGe simply requires exclusive access rights to six DRAM rows per bank, consisting of the two rows containing the RNG cells and each row's two physically-adjacent DRAM rows containing the chosen data pattern.[8] This results in an insignificant 0.018% DRAM storage overhead cost.

To evaluate D-RaNGe's effect on the DRAM access latency of regular memory requests, we present one implementation of D-RaNGe. For a single DRAM channel, which is the granularity at which DRAM timing parameters are applied, D-RaNGe can alternate between using a reduced t_{RCD} and the default t_{RCD}. When using a reduced t_{RCD}, D-RaNGe generates random numbers across every bank in the channel. On the other hand, when using the default t_{RCD}, memory requests from running applications are serviced to ensure application progress. The length of these time intervals (with default/reduced t_{RCD}) can both be adjusted according to the applications' random number generation requirements. Overall, D-RaNGe provides significant flexibility in trading off its system overhead with its TRNG throughput. However, it is up to the system designer to use and exploit the flexibility for their requirements. To show the potential throughput of D-RaNGe without impacting concurrently-running applications, we run simulations with the SPEC CPU2006 [3] workloads, and calculate the idle DRAM bandwidth available that we can use to issue D-RaNGe commands. We find that, across all workloads, we can obtain an average (maximum, minimum) random-value throughput of 83.1 (98.3, 49.1) Mb/s with *no* significant impact on overall system performance.

Low Energy Consumption. To evaluate the energy consumption of D-RaNGe, we use DRAMPower [1] to analyze the output traces of Ramulator [2, 76] when DRAM is (1) generating random numbers (Algorithm 2), and (2) idling and not servicing memory requests. We subtract quantity (2) from (1) to obtain the estimated energy consumption of D-RaNGe. We then divide the value by the total number of random bits found during execution and find that, on average, D-RaNGe finds random bits at the cost of 4.4 nJ/bit.

8. Comparison with Prior DRAM TRNGs

To our knowledge, this paper provides the highest-throughput TRNG *for commodity DRAM devices* by exploiting activation failures as a sampling mechanism for observing entropy in DRAM cells. There are a number of proposals to construct TRNGs using commodity DRAM devices, which we summarize in Table 2 based on their entropy sources. In this section, we compare each of these works with D-RaNGe. We show how D-RaNGe fulfills the six key properties of an ideal TRNG (Section 3) better than any prior DRAM-based TRNG proposal. We group our comparisons by the entropy source of each prior DRAM-based TRNG proposal.

8.1. DRAM Command Scheduling

Prior work [116] proposes using non-determinism in DRAM command scheduling for true random number generation. In particular, since pending access commands contend with regular refresh operations, the latency of a DRAM access is hard to predict and is useful for random number generation.

Unfortunately, this method fails to satisfy two important properties of an ideal TRNG. First, it harvests random numbers from the instruction and DRAM command scheduling decisions made by the processor and memory controller, which does *not* constitute a fully non-deterministic entropy source. Since the quality of the harvested random numbers depends directly on the quality of the processor and memory controller implementations, the entropy source is visible to and potentially modifiable by an adversary (e.g., by simultaneously running a memory-intensive workload on another processor core [104]). Therefore, this method does not meet our design goals as it does not securely generate random numbers.

Second, although this technique has a higher throughput than those based on DRAM data retention (Table 2), D-RaNGe still outperforms this method in terms of throughput by 211x (maximum) and 128x (average) because a single byte of random data requires a *significant* amount of time to generate. Even if we scale the throughput results provided by [116] to a modern day system (e.g., 5GHz processor, 4 DRAM channels[9]), the theoretical maximum throughput of Pyo et al.'s approach[10] is *only* 3.40Mb/s as compared with the maximum (average) throughput of 717.4Mb/s (435.7Mb/s) for D-RaNGe. To calculate the latency of generating random values, we assume the same system configuration with [116]'s claimed number of cycles 45000 to generate random bits. To provide 64 bits of random data, [116] takes 18μs, which is significantly higher than D-RaNGe's minimum/maximum latency of 100ns/960ns. Energy consumption for [116] depends heavily on the entire system that it is running on, so we do not compare against this metric.

8.2. DRAM Data Retention

Prior works [65, 141] propose using DRAM data retention failures to generate random numbers. Unfortunately, this approach is *inherently too slow* for high-throughput operation due to the long wait times required to induce data retention failures in DRAM. While the failure rate can be increased by increasing the operating temperature, a wait time on the order of seconds is required to induce enough failures [66, 72, 91, 112, 117] to achieve high-throughput random number generation, which is orders of magnitude slower than D-RaNGe.

Sutar et al. [141] report that they are able to generate 256-bit random numbers using a hashing algorithm (e.g., SHA-256) on a 4 *MiB* DRAM block that contains data retention errors resulting from having disabled DRAM refresh for 40 seconds. Optimistically assuming a large DRAM capacity of 32 *GiB* and ignoring the time required to read out and hash the erroneous data, a waiting time of 40 seconds to induce data retention errors allows for an estimated maximum random number throughput of 0.05 *Mb/s*. This throughput is already far smaller than D-RaNGe's measured maximum throughput of 717.4Mb/s, and it would decrease linearly with DRAM capacity. Even if we were able to induce a large number of data retention errors by waiting only 1 second, the maximum random number generation throughput would be 2 *Mb/s*, i.e., orders of magnitude smaller than that of D-RaNGe.

Because [141] requires a wait time of 40 seconds before producing any random values, its latency for random number generation is extremely high (40s). In contrast, D-RaNGe

[8] As in prior work [14, 73, 106], we argue that manufacturers can disclose which rows are physically adjacent to each other.

[9] The authors do not provide their DRAM configuration, so we optimistically assume that they evaluate their proposal using one DRAM channel. We also assume that by utilizing 4 DRAM channels, the authors can harvest four times the entropy, which gives the benefit of the doubt to [116].

[10] We base our estimations on [116]'s claim that they can harvest one byte of random data every 45000 cycles. However, using these numbers along with the authors' stated processor configuration (i.e., 2.8GHz) leads to a discrepancy between our calculated maximum throughput ($\approx$ 0.5Mb/s) and that reported in [116] ($\approx$ 5Mb/s). We believe our estimation methodology and calculations are sound. In our work, we compare D-RaNGe's peak throughput against that of [116] using a more modern system configuration (i.e., 5GHz processor, 4 DRAM channels) than used in the original work, which gives the benefit of the doubt to [116].

Proposal	Year	Entropy Source	True Random	Streaming Capable	64-bit TRNG Latency	Energy Consumption	Peak Throughput
Pyo+ [116]	2009	Command Schedule	✗	✓	$18\mu s$	N/A	$3.40Mb/s$
Keller+ [65]	2014	Data Retention	✓	✓	$40s$	$6.8m\mathcal{J}/bit$	$0.05Mb/s$
Tehranipoor+ [144]	2016	Startup Values	✓	✗	$> 60ns$ (optimistic)	$> 245.9p\mathcal{J}/bit$ (optimistic)	N/A
Sutar+ [141]	2018	Data Retention	✓	✓	$40s$	$6.8m\mathcal{J}/bit$	$0.05Mb/s$
D-RaNGe	2018	Activation Failures	✓	✓	$100ns < x < 960ns$	$4.4n\mathcal{J}/bit$	$717.4Mb/s$

Table 2: Comparison to previous DRAM-based TRNG proposals.

can produce random values very quickly since it generates random values potentially with each DRAM access (10s of nanoseconds). D-RaNGe therefore has a latency many orders of magnitude lower than Sutar et al.'s mechanism [141].

We estimate the energy consumption of retention-time based TRNG mechanisms with Ramulator [2, 76] and DRAMPower [1, 25]. We model first writing data to a 4MiB DRAM region (to constrain the energy consumption estimate to the region of interest), waiting for 40 seconds, and then reading from that region. We then divide the energy consumption of these operations by the number of bits found (256 bits). We find that the energy consumption is around $6.8m\mathcal{J}$ per bit, which is orders of magnitude more costly than that of D-RaNGe, which provides random numbers at $4.4n\mathcal{J}$ per bit.

8.3. DRAM Startup Values

Prior work [39, 144] proposes using DRAM startup values as random numbers. Unfortunately, this method is unsuitable for continuous high-throughput operation since it requires a DRAM power cycle in order to obtain random data. We are unable to accurately model the latency of this mechanism since it relies on the startup time of DRAM (i.e., bus frequency calibration, temperature calibration, timing register initialization [60]). This heavily depends on the implementation of the system and the DRAM device in use. Ignoring these components, we estimate the throughput of generating random numbers using startup values by taking into account only the latency of a single DRAM read (*after* all initialization is complete), which is $60ns$. We model energy consumption ignoring the initialization phase as well, by modeling the energy to read a MiB of DRAM and dividing that quantity by [144]'s claimed number of random bits found in that region (420Kbit). Based on this calculation, we estimate energy consumption as $245.9p\mathcal{J}$ per bit. While the energy consumption of [144] is smaller than the energy cost of D-RaNGe, we note that our energy estimation for [144] does *not* account for the energy consumption required for initializing DRAM to be able to read out the random values. Additionally, [144] requires a full system reboot which is often impractical for applications and for effectively providing a *steady stream of random values*. [39] suffers from the same issues since it uses the same mechanism as [144] to generate random numbers and is strictly worse since [39] results in $31.8x$ less entropy.

8.4. Combining DRAM-based TRNGs

We note that D-RaNGe's method for sampling random values from DRAM is entirely distinct from prior DRAM-based TRNGs that we have discussed in this section. This makes it possible to combine D-RaNGe with prior work to produce random values at an even higher throughput.

9. Other Related Works

In this work, we focus on the design of a DRAM-based hardware mechanism to implement a TRNG, which makes the focus of our work orthogonal to those that design PRNGs. In contrast to prior DRAM-based TRNGs discussed in Section 8, we propose using *activation failures* as an entropy source. Prior works characterize activation failures in order to exploit the resulting error patterns for overall DRAM latency reduction [27, 71, 84, 87] and to implement physical unclonable functions (PUFs) [72]. However, none of these works measure the randomness inherent in activation failures or propose using them to generate random numbers.

Many TRNG designs have been proposed that exploit sources of entropy that are *not* based on DRAM. Unfortunately, these proposals either 1) require custom hardware modifications that preclude their application to commodity devices, or 2) do not sustain continuous (i.e., constant-rate) high-throughput operation. We briefly discuss different entropy sources with examples.

Flash Memory Read Noise. Prior proposals use random telegraph noise in flash memory devices as an entropy source (up to 1 Mbit/s) [119, 153]. Unfortunately, flash memory is orders of magnitude slower than DRAM, making flash unsuitable for high-throughput and low-latency operation.

SRAM-based Designs. SRAM-based TRNG designs exploit randomness in startup values [55, 56, 151]. Unfortunately, these proposals are unsuitable for continuous, high-throughput operation since they require a power cycle.

GPU- and FPGA-Based Designs. Several works harvest random numbers from GPU-based (up to 447.83 Mbit/s) [24, 143, 149] and FPGA-based (up to 12.5 Mbit/s) [33, 54, 96, 155] entropy sources. These proposals do not require modifications to commodity GPUs or FPGAs. Yet, GPUs and FPGAs are not as prevalent as DRAM in commodity devices today.

Custom Hardware. Various works propose TRNGs based in part or fully on non-determinism provided by custom hardware designs (with TRNG throughput up to 2.4 Gbit/s) [6, 16, 22, 23, 56, 57, 77, 101, 111, 114, 134, 146, 158]. Unfortunately, the need for custom hardware limits the widespread use of such proposals in commodity hardware devices (today).

10. Conclusion

We propose D-RaNGe, a mechanism for extracting true random numbers with high throughput from unmodified commodity DRAM devices on any system that allows manipulation of DRAM timing parameters in the memory controller. D-RaNGe harvests fully non-deterministic random numbers from DRAM row activation failures, which are bit errors induced by intentionally accessing DRAM with lower latency than required for correct row activation. Our TRNG is based on two key observations: 1) activation failures can be induced quickly and 2) repeatedly accessing certain DRAM cells with reduced activation latency results in reading true random data. We validate the quality of our TRNG with the commonly-used NIST statistical test suite for randomness. Our evaluations show that D-RaNGe significantly outperforms the previous highest-throughput DRAM-based TRNG by up to 211x (128x on average). We conclude that DRAM row activation failures can be effectively exploited to efficiently generate true random numbers with high throughput on a wide range of devices that use commodity DRAM chips.

Acknowledgments

We thank the anonymous reviewers of HPCA 2019 and MICRO 2018 for feedback and the SAFARI group members for feedback and the stimulating intellectual environment they provide.

References

[1] "DRAMPower Source Code," https://github.com/tukl-msd/DRAMPower.
[2] "Ramulator Source Code," https://github.com/CMU-SAFARI/ramulator.
[3] "Standard Performance Evaluation Corporation," http://www.spec.org/cpu2006.
[4] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.
[5] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "PIM-Enabled Instructions: a Low-overhead, Locality-aware Processing-in-Memory Architecture," in *ISCA*, 2015.
[6] T. Amaki, M. Hashimoto, and T. Onoye, "An Oscillator-based True Random Number Generator with Process and Temperature Tolerance," in *DAC*, 2015.
[7] AMD, "AMD Opteron 4300 Series Processors," 2012.
[8] AMD, "BKDG for AMD Family 16h Models 00h-0Fh Processors," 2013.
[9] ARM, "ARM CoreLink DMC-520 Dynamic Memory Controller Technical Reference Manual," 2016.
[10] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu, "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," in *ISCA*, 2012.
[11] O. O. Babarinsa and S. Idreos, "JAFAR: Near-Data Processing for Databases," in *SIGMOD*, 2015.
[12] S. Bae, Y. Kim, Y. Park, and C. Kim, "3-Gb/s High-speed True Random Number Generator using Common-mode Operating Comparator and Sampling Uncertainty of D Flip-flop," in *JSSC*, 2017.
[13] V. Bagini and M. Bucci, "A Design of Reliable True Random Number Generator for Cryptographic Applications," in *CHES*, 1999.
[14] K. S. Bains, J. B. Halbert, S. Sah, and Z. Greenfield, "Method, apparatus and system for providing a memory refresh," US Patent 9,030,903. May 12 2015.
[15] M. Barangi, J. S. Chang, and P. Mazumder, "Straintronics-Based True Random Number Generator for High-Speed and Energy-Limited Applications," in *IEEE Trans. Magn*, 2016.
[16] M. Bhargava, K. Sheikh, and K. Mai, "Robust True Random Number Generator using Hot-carrier Injection Balanced Metastable Sense Amplifiers," in *HOST*, 2015.
[17] I. Bhati, Z. Chishti, S.-L. Lu, and B. Jacob, "Flexible Auto-Refresh: Enabling Scalable and Energy-Efficient DRAM Refresh Reductions," in *ISCA*, 2015.
[18] L. Blum, M. Blum, and M. Shub, "A Simple Unpredictable pseudo-random Number Generator," in *SIAM Journal on Computing*, 1986.
[19] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," in *ASPLOS*, 2018.
[20] A. Boroumand, S. Ghose, B. Lucia, K. Hsieh, K. Malladi, H. Zheng, and O. Mutlu, "LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory," in *CAL*, 2017.
[21] R. Botha, "The Development of a Hardware Random Number Generator for Gamma-ray Astronomy," Ph.D. dissertation, North-West University, 2005.
[22] R. Brederlow, R. Prakash, C. Paulus, and R. Thewes, "A Low-power True Random Number Generator using Random Telegraph Noise of Single Oxide-traps," in *ISSCC*, 2006.
[23] M. Bucci, L. Germani, R. Luzzi, A. Trifiletti, and M. Varanonuovo, "A High-speed Oscillator-based Truly Random Number Source for Cryptographic Applications on a Smart Card IC," in *TC*, 2003.
[24] J. J. M. Chan, B. Sharma, J. Lv, G. Thomas, R. Thulasiram, and P. Thulasiraman, "True Random Number Generator using GPUs and Histogram Equalization Techniques," in *HPCC*, 2011.
[25] K. Chandrasekar, B. Akesson, and K. Goossens, "Improved Power Modeling of DDR SDRAMs," in *DSD*, 2011.
[26] K. K. Chang, "Understanding and Improving Latency of DRAM-Based Memory Systems," Ph.D. dissertation, Carnegie Mellon University, 2017.
[27] K. K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *SIGMETRICS*, 2016.
[28] K. K. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," in *HPCA*, 2014.
[29] K. K. Chang, P. J. Nair, D. Lee, S. Ghose, M. K. Qureshi, and O. Mutlu, "Low-cost Inter-linked Subarrays (LISA): Enabling Fast Inter-subarray Data Movement in DRAM," in *HPCA*, 2016.
[30] K. K. Chang, A. G. Yağlikçi, S. Ghose, A. Agrawal, N. Chatterjee, A. Kashyap, D. Lee, M. O'Connor, H. Hassan, and O. Mutlu, "Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms," in *SIGMETRICS*, 2017.
[31] A. Cherkaoui, V. Fischer, L. Fesquet, and A. Aubert, "A Very High Speed True Random Number Generator with Entropy Assessment," in *CHES*, 2013.
[32] P. Chevalier, C. Menard, and B. Dorval, "Random Number Generator," US Patent 3,790,768. 1974.
[33] P. P. Chu and R. E. Jones, "Design Techniques of FPGA Based Random Number Generator," in *MAPLD*, 1999.
[34] P. J. Clarke, R. J. Collins, P. A. Hiskett, P. D. Townsend, and G. S. Buller, "Robust Gigahertz Fiber Quantum Key Distribution," *Applied Physics Letters*, 2011.
[35] H. Corrigan-Gibbs, W. Mu, D. Boneh, and B. Ford, "Ensuring High-quality Randomness in Cryptographic Key Generation," in *CCS*, 2013.
[36] L. Dorrendorf, Z. Gutterman, and B. Pinkas, "Cryptanalysis of the Windows Random Number Generator," in *CCS*, 2007.
[37] M. Drutarovsky and P. Galajda, "A Robust Chaos-based True Random Number Generator Embedded in Reconfigurable Switched-Capacitor Hardware," in *Radioelektronika*, 2007.
[38] D. Eastlake and P. Jones, "US Secure Hash Algorithm 1 (SHA1)," Tech. Rep., 2001.
[39] C. Eckert, F. Tehranipoor, and J. A. Chandy, "DRNG: DRAM-based Random Number Generation Using its Startup Value Behavior," in *MWSCAS*, 2017.
[40] G. Fabron, "RAM Overclocking Guide: How (and Why) to Tweak Your Memory," https://www.tomshardware.com/reviews/ram-overclocking-guide,4693.html.
[41] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules," in *HPCA*, 2015.
[42] V. Fischer, M. Drutarovskỳ, M. Šimka, and N. Bochard, "High performance true random number generator in Altera Stratix FPLDs," in *FPL*, 2004.
[43] M. Gao, G. Ayers, and C. Kozyrakis, "Practical Near-Data Processing for In-Memory Analytics Frameworks," in *PACT*, 2015.
[44] M. Gao and C. Kozyrakis, "HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing," in *HPCA*, 2016.
[45] S. Ghose, K. Hsieh, A. Boroumand, R. Ausavarungnirun, and O. Mutlu, "Enabling the Adoption of Processing-in-Memory: Challenges, Mechanisms, Future Research Directions," *arXiv preprint arXiv:1802.00320*, 2018.
[46] S. Ghose, G. Yaglikci, R. Gupta, D. Lee, K. Kudrolli, W. Liu, H. Hassan, K. Chang, N. Chatterjee, A. Agrawal, M. O'Connor, and O. Mutlu, "What Your DRAM Power Models Are Not Telling You: Lessons from a Detailed Experimental Study," in *SIGMETRICS*, 2018.
[47] Z. Gutterman, B. Pinkas, and T. Reinman, "Analysis of the Linux Random Number Generator," in *SP*, 2006.
[48] T. Gyorfi, O. Cret, and A. Suciu, "High Performance True Random Number Generator Based on FPGA Block RAMs," in *IPDPS*, 2009.
[49] M. Hamburg, P. Kocher, and M. E. Marson, "Analysis of Intel's Ivy Bridge Digital Random Number Generator," *www.cryptography.com/public/pdf/Intel_TRNG_Report_20120312.pdf*, 2012.
[50] M. S. Hashemian, B. Singh, F. Wolff, D. Weyer, S. Clay, and C. Papachristou, "A Robust Authentication Methodology Using Physically Unclonable Functions in DRAM Arrays," in *DATE*, 2015.
[51] H. Hassan, G. Pekhimenko, N. Vijaykumar, V. Seshadri, D. Lee, O. Ergin, and O. Mutlu, "ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality," in *HPCA*, 2016.
[52] H. Hassan, N. Vijaykumar, S. Khan, S. Ghose, K. Chang, G. Pekhimenko, D. Lee, O. Ergin, and O. Mutlu, "SoftMC: A Flexible and Practical Open-source Infrastructure for Enabling Experimental DRAM Studies," in *HPCA*, 2017.
[53] S. M. Hassan, S. Yalamanchili, and S. Mukhopadhyay, "Near Data Processing: Impact and Optimization of 3D Memory System Architecture on the Uncore," in *MEMSYS*, 2015.
[54] H. Hata and S. Ichikawa, "FPGA Implementation of Metastability-based True Random Number Generator," *IEICE Trans. Inf. Syst.*, 2012.
[55] D. E. Holcomb, W. P. Burleson, and K. Fu, "Initial SRAM State as a Fingerprint and Source of True Random Numbers for RFID Tags," in *RFID*, 2007.
[56] D. E. Holcomb, W. P. Burleson, and K. Fu, "Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers," in *TC*, 2009.
[57] J. Holleman, S. Bridges, B. P. Otis, and C. Diorio, "A 3mu W CMOS True Random Number Generator with Adaptive Floating-Gate Offset Cancellation," *JSSC*, 2008.
[58] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *ISCA*, 2016.
[59] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation," in *ICCD*, 2016.
[60] S. Hynix, "DDR4 SDRAM Device Operation."
[61] Intel, "Intel Architecture Software Developer's Manual," 2018.
[62] JEDEC, *Double Data Rate 3 (DDR3) SDRAM Specification*, 2012.
[63] JEDEC, *Low Power Double Data Rate 4 (LPDDR4) SDRAM Specification*, 2014.
[64] B. Jun and P. Kocher, "The Intel Random Number Generator," *Cryptography Research Inc. white paper*, 1999.
[65] C. Keller, F. Gurkaynak, H. Kaeslin, and N. Felber, "Dynamic Memory-based Physically Unclonable Function for the Generation of Unique Identifiers and True Random Numbers," in *ISCAS*, 2014.
[66] S. Khan, D. Lee, Y. Kim, A. R. Alameldeen, C. Wilkerson, and O. Mutlu, "The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study," in *SIGMETRICS*, 2014.
[67] S. Khan, D. Lee, and O. Mutlu, "PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM," in *DSN*, 2016.
[68] S. Khan, C. Wilkerson, Z. Wang, A. R. Alameldeen, D. Lee, and O. Mutlu, "Detecting and Mitigating Data-Dependent DRAM Failures by Exploiting Current Memory Content," in *MICRO*, 2017.
[69] J. Kim, T. Ahmed, H. Nili, N. D. Truong, J. Yang, D. S. Jeong, S. Sriram, D. C. Ranasinghe, and O. Kavehei, "Nano-Intrinsic True Random Number Generation," *arXiv preprint arXiv:1701.06020*, 2017.
[70] J. S. Kim, D. S. Cali, H. Xin, D. Lee, S. Ghose, M. Alser, H. Hassan, O. Ergin, C. Alkan, and O. Mutlu, "GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-memory Technologies," *BMC Genomics*, 2018.
[71] J. S. Kim, M. Patel, H. Hassan, and O. Mutlu, "Solar-DRAM: Reducing DRAM Access Latency by Exploiting the Variation in Local Bitlines," in *ICCD*, 2018.
[72] J. S. Kim, M. Patel, H. Hassan, and O. Mutlu, "The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern Commodity DRAM Devices," in *HPCA*, 2018.
[73] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.
[74] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.
[75] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A Case for Exploiting Subarray-level Parallelism (SALP) in DRAM," in *ISCA*, 2012.
[76] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A Fast and Extensible DRAM Simulator," in *CAL*, 2016.
[77] D. Kinniment and E. Chester, "Design of an On-chip Random Number Generator using Metastability," in *ESSCIRC*, 2002.
[78] D. E. Knuth, "The art of computer programming, 2: Seminumerical agorithms, addision wesley," *Reading, MA*, 1998.
[79] Ç. K. Koç, "About Cryptographic Engineering," in *Cryptographic Engineering*, 2009.
[80] S. H. Kwok and E. Y. Lam, "FPGA-based High-speed True Random Number Generator for Cryptographic Applications," in *TENCON*, 2006.
[81] S. Kwok, Y. Ee, G. Chew, K. Zheng, K. Khoo, and C. Tan, "A Comparison of Post-processing Techniques for Biased Random Number Generators," in *WISTP*, 2011.
[82] Q. Labs, "Random Number Generators White Paper," 2015.

[83] P. Lacharme, A. Rock, V. Strubel, and M. Videau, "The Linux Pseudorandom Number Generator Revisited," 2012.

[84] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu, "Adaptive-latency DRAM: Optimizing DRAM Timing for the Common-case," in *HPCA*, 2015.

[85] D. Lee, "Reducing DRAM Latency at Low Cost by Exploiting Heterogeneity," Ph.D. dissertation, Carnegie Mellon University, 2016.

[86] D. Lee, S. Ghose, G. Pekhimenko, S. Khan, and O. Mutlu, "Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost," in *TACO*, 2016.

[87] D. Lee, S. Khan, L. Subramanian, S. Ghose, R. Ausavarungnirun, G. Pekhimenko, V. Seshadri, and O. Mutlu, "Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms," in *SIGMETRICS*, 2017.

[88] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," in *HPCA*, 2013.

[89] D. Lee, L. Subramanian, R. Ausavarungnirun, J. Choi, and O. Mutlu, "Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM," in *PACT*, 2015.

[90] J. Lin, W. Yu, N. Zhang, X. Yang, H. Zhang, and W. Zhao, "A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications," in *IoT*, 2017.

[91] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ISCA*, 2013.

[92] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-Aware Intelligent DRAM Refresh," in *ISCA*, 2012.

[93] Z. Liu, I. Calciu, M. Herlihy, and O. Mutlu, "Concurrent Data Structures for Near-Memory Computing," in *SPAA*, 2017.

[94] X. Lu, L. Zhang, Y. Wang, W. Chen, D. Huang, D. Li, S. Wang, D. He, Z. Yin, Y. Zhou *et al.*, "FPGA Based Digital Phase-coding Quantum Key Distribution System," *Science China Physics, Mechanics & Astronomy*, 2015.

[95] X. Ma, X. Yuan, Z. Cao, B. Qi, and Z. Zhang, "Quantum Random Number Generation," *Quantum Inf.*, 2016.

[96] M. Majzoobi, F. Koushanfar, and S. Devadas, "FPGA-based True Random Number Generation using Circuit Metastability with Adaptive Feedback Control," in *CHES*, 2011.

[97] G. Marsaglia, "The Marsaglia Random Number CDROM Including the Diehard Battery of Tests of Randomness," 2008, http://www.stat.fsu.edu/pub/diehard/.

[98] G. Marsaglia *et al.*, "Xorshift RNGs," *Journal of Statistical Software*, 2003.

[99] K. Marton and A. Suciu, "On the Interpretation of Results from the NIST Statistical Test Suite," *Science and Technology*, 2015.

[100] M. Mascagni and A. Srinivasan, "Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation," in *TOMS*, 2000.

[101] S. K. Mathew, S. Srinivasan, M. A. Anders, H. Kaul, S. K. Hsu, F. Sheikh, A. Agarwal, S. Satpathy, and R. K. Krishnamurthy, "2.4 Gbps, 7 mW All-digital PVT-variation Tolerant True Random Number Generator for 45 nm CMOS High-performance Microprocessors," in *JSSC*, 2012.

[102] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator," in *TOMACS*, 1998.

[103] A. Morad, L. Yavits, and R. Ginosar, "GP-SIMD Processing-in-Memory," in *TACO*, 2015.

[104] T. Moscibroda and O. Mutlu, "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems," in *USENIX Security*, 2007.

[105] J. Mukundan, H. Hunter, K.-h. Kim, J. Stuecheli, and J. F. Martínez, "Understanding and Mitigating Refresh Overheads in High-density DDR4 DRAM Systems," in *ISCA*, 2013.

[106] O. Mutlu, "The RowHammer Problem and Other Issues we may Face as Memory Becomes Denser," in *DATE*, 2017.

[107] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.

[108] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enabling High-performance And Fair Shared Memory Controllers," in *ISCA*, 2008.

[109] P. J. Nair, V. Sridharan, and M. K. Qureshi, "XED: Exposing On-Die Error Detection Information for Strong Memory Reliability," in *ISCA*, 2016.

[110] L. Ning, J. Ding, B. Chuang, and Z. Xuecheng, "Design and Validation of High Speed True Random Number Generators Based on Prime-length Ring Oscillators," *The Journal of China Universities of Posts and Telecommunications*, 2015.

[111] F. Pareschi, G. Setti, and R. Rovatti, "A Fast Chaos-based True Random Number Generator for Cryptographic Applications," in *ESSCIRC*, 2006.

[112] M. Patel, J. S. Kim, and O. Mutlu, "The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions," in *ISCA*, 2017.

[113] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, "Scheduling Techniques for GPU Architectures with Processing-in-Memory Capabilities," in *PACT*, 2016.

[114] C. S. Petrie and J. A. Connelly, "A Noise-based IC Random Number Generator for Applications in Cryptography," in *Trans. Circuits Syst. I*, 2000.

[115] Ponemon Institute LLC, "Study on Mobile and IoT Application Security," 2017.

[116] C. Pyo, S. Pae, and G. Lee, "DRAM as Source of Randomness," in *IET*, 2009.

[117] M. K. Qureshi, D. Kim, S. Khan, P. J. Nair, and O. Mutlu, "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems," in *DSN*, 2015.

[118] M. T. Rahman, K. Xiao, D. Forte, X. Zhang, J. Shi, and M. Tehranipoor, "TI-TRNG: Technology Independent True Random Number Generator," in *DAC*, 2014.

[119] B. Ray and A. Milenković, "True Random Number Generation Using Read Noise of Flash Memory Cells," in *IEEE Trans. on Electron Devices*, 2018.

[120] R. Rivest, "The MD5 Message-Digest Algorithm," in *RFC*, 1992.

[121] A. Röck, *Pseudorandom Number Generators for Cryptographic Applications*, 2005.

[122] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker, "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications," Booz-Allen and Hamilton Inc Mclean Va, Tech. Rep., 2001.

[123] Samsung, "S5P4418 Application Processor Revision 0.10," 2014.

[124] W. Schindler and W. Killmann, "Evaluation Criteria for True (Physical) Random Number Generators Used in Cryptographic Applications," in *CHES*, 2002.

[125] V. Seshadri, "Simple DRAM and Virtual Memory Abstractions to Enable Highly Efficient Memory Systems," Ph.D. dissertation, Carnegie Mellon University, 2016.

[126] V. Seshadri, K. Hsieh, A. Boroumand, D. Lee, M. Kozuch, O. Mutlu, P. Gibbons, and T. Mowry, "Fast Bulk Bitwise AND and OR in DRAM," *CAL*, 2015.

[127] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. Mowry, "RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization," in *MICRO*, 2013.

[128] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," in *MICRO*, 2017.

[129] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Gather-scatter DRAM: In-DRAM Address Translation to Improve the Spatial Locality of Non-unit Strided Accesses," in *MICRO*, 2015.

[130] V. Seshadri and O. Mutlu, "Simple Operations in Memory to Reduce Data Movement," in *Advances in Computers*, 2017.

[131] C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal*, 1948.

[132] SoftMC Source Code, https://github.com/CMU-SAFARI/SoftMC.

[133] G. L. Steele Jr, D. Lea, and C. H. Flood, "Fast Splittable Pseudorandom Number Generators," in *OOPSLA*, 2014.

[134] A. Stefanov, N. Gisin, O. Guinnard, L. Guinnard, and H. Zbinden, "Optical Quantum Random Number Generator," in *J. Mod. Opt*, 2000.

[135] M. Stipčević and Ç. K. Koç, "True Random Number Generators," in *Open Problems in Mathematics and Computational Science*, 2014.

[136] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling," in *TPDS*, 2016.

[137] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-application Interference at Shared Caches and Main Memory," in *MICRO*, 2015.

[138] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in *HPCA*, 2013.

[139] B. Sunar, W. J. Martin, and D. R. Stinson, "A Provably Secure True Random Number Generator with Built-in Tolerance to Active Attacks," in *TC*, 2007.

[140] Z. Sura, A. Jacob, T. Chen, B. Rosenburg, O. Sallenave, C. Bertolli, S. Antao, J. Brunheroto, Y. Park, K. O'Brien *et al.*, "Data Access Optimization in a Processing-in-Memory System," in *CF*, 2015.

[141] S. Sutar, A. Raha, D. Kulkarni, R. Shorey, J. Tew, and V. Raghunathan, "D-PUF: An Intrinsically Reconfigurable DRAM PUF for Device Authentication and Random Number Generation," in *TECS*, 2018.

[142] S. Tao and E. Dubrova, "TVL-TRNG: Sub-Microwatt True Random Number Generator Exploiting Metastability in Ternary Valued Latches," in *ISMVL*, 2017.

[143] J. S. Teh, A. Samsudin, M. Al-Mazrooie, and A. Akhavan, "GPUs and Chaos: A New True Random Number Generator," in *Nonlinear Dynamics*, 2015.

[144] F. Tehranipoor, W. Yan, and J. A. Chandy, "Robust Hardware True Random Number Generators using DRAM Remanence Effects," in *HOST*, 2016.

[145] H. C. v. Tilborg and S. Jajodia, "Encyclopedia of Cryptography and Security," 2011.

[146] C. Tokunaga, D. Blaauw, and T. Mudge, "True Random Number Generator with a Metastability-based Quality Control," in *JSSC*, 2008.

[147] K. H. Tsoi, K. H. Leung, and P. H. W. Leong, "High Performance Physical Random Number Generator," in *IET computers & digital techniques*, 2007.

[148] K. H. Tsoi, K. Leung, and P. H. W. Leong, "Compact FPGA-based True and Pseudo Random Number Generators," in *FCCM*, 2003.

[149] S. Tzeng and L. Wei, "Parallel White Noise Generation on a GPU via Cryptographic Hash," in *I3D*, 2008.

[150] H. Usui, L. Subramanian, K. K. Chang, and O. Mutlu, "DASH: Deadline-Aware High-performance Memory Scheduler for Heterogeneous Systems fith Hardware Accelerators," in *TACO*, 2016.

[151] V. van der Leest, E. van der Sluis, G.-J. Schrijen, P. Tuyls, and H. Handschuh, "Efficient Implementation of True Random Number Generator Based on SRAM PUFs," in *Cryptography and Security: From Theory to Applications*, 2012.

[152] V. von Kaenel and T. Takayanagi, "Dual True Random Number Generators for Cryptographic Applications Embedded on a 200 Million Device Dual CPU SOC," in *CICC*, 2007.

[153] Y. Wang, W. Yu, S. Wu, G. Malysa, G. E. Suh, and E. C. Kan, "Flash Memory for Ubiquitous Hardware Security Functions: True Random Number Generation and Device Fingerprints," in *SP*, 2012.

[154] Y. Wang, C. Hui, C. Liu, and C. Xu, "Theory and Implementation of a Very High Throughput True Random Number Generator in Field Programmable Gate Array," *Review of Scientific Instruments*, 2016.

[155] P. Z. Wieczorek, "An FPGA Implementation of the Resolve Time-based True Random Number Generator with Quality Control."

[156] G. Wolrich, D. Bernstein, D. Cutter, C. Dolan, and M. J. Adiletta, "Mapping Requests from a Processing Unit That Uses Memory-Mapped Input-Output Space," US Patent 6,694,380. 2004.

[157] D. S. Yaney, C. Lu, R. A. Kohler, M. J. Kelly, and J. T. Nelson, "A Meta-stable Leakage Phenomenon in DRAM Charge Storage-Variable Hold Time," in *IEDM*, 1987.

[158] K. Yang, D. Blaauw, and D. Sylvester, "An All-digital Edge Racing True Random Number Generator Robust Against PVT Variations," in *JSSC*, 2016.

[159] K. Yang, D. Fick, M. B. Henry, Y. Lee, D. Blaauw, and D. Sylvester, "16.3 A 23Mb/s 23pJ/b Fully Synthesized True-random-number Generator in 28nm and 65nm CMOS," in *ISSCC*, 2014.

[160] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-Oriented Programmable Processing in Memory," in *HPDC*, 2014.

[161] T. Zhang, K. Chen, C. Xu, G. Sun, T. Wang, and Y. Xie, "Half-DRAM: A High-bandwidth and Low-power DRAM Architecture from the Rethinking of Fine-grained Activation," in *ISCA*, 2014.

[162] T. Zhang, M. Yin, C. Xu, X. Lu, X. Sun, Y. Yang, and R. Huang, "High-speed True Random Number Generation Based on Paired Memristors for Security Electronics," *Nanotechnology*, 2017.

[163] X. Zhang, Y. Nie, H. Zhou, H. Liang, X. Ma, J. Zhang, and J. Pan, "68 Gbps Quantum Random Number Generation by Measuring Laser Phase Fluctuations," in *Review of Scientific Instruments*, 2015.