RingRAM: A Unified Hardware Security Primitive for IoT Devices that Gets Better with Age

Michael Moukarzel
mamoukar@vt.edu
Virginia Tech
Blacksburg, Virginia, USA

Matthew Hicks
mdhicks2@vt.edu
Virginia Tech
Blacksburg, Virginia, USA

ABSTRACT

As security grows in importance, system designers turn to hardware support for security. Hardware's unique properties enable functionality and performance levels not available with software alone. One unique property of hardware is non-determinism. Unlike software, which is inherently deterministic (e.g., the same inputs produce the same outputs), hardware encompasses an abundance of non-determinism; non-determinism born out of manufacturing and operational chaos. While hardware designers focus on hiding the effects of such chaos behind voltage and clock frequency guard bands, security practitioners embrace the chaos as a source of randomness.

We propose a single hardware security primitive composed of basic circuit elements that harnesses both manufacturing and operational chaos to serve as the foundation for both a true random-number generator and a physical unclonable function suitable for deployment in resource-constrained Internet-of-Things (IoT) devices. Our primitive RingRAM leverages the observation that, while existing hardware security primitives have limitations that prevent deployment, they can be merged to form a hardware security primitive that has all of the benefits, but none of the drawbacks. We show how RingRAM's reliance on simple circuit elements enables universal implementation using discrete components, on an FPGA, and as an ASIC. We then design RingRAM tuning knobs that allow designers to increase entropy, decrease noise, and eliminate off-chip post-processing. We validate RingRAM, showing that it serves as a superior true random-number generator and physical unclonable function—robust against aging and thermal attacks. Finally, to show how RingRAM increases IoT system security, we provide two Linux-based use cases on top of a RISC-V System-on-Chip.

ACM Reference Format:
Michael Moukarzel and Matthew Hicks. 2021. RingRAM: A Unified Hardware Security Primitive for IoT Devices that Gets Better with Age. In *Annual Computer Security Applications Conference (ACSAC '21), December 6–10, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3485832.3485905

ACSAC '21, December 6–10, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8579-4/21/12…$15.00
https://doi.org/10.1145/3485832.3485905

1 INTRODUCTION

Random numbers are the foundation for the secure systems that society depends on. Since all cryptographic algorithms are deterministic by design, when given the same inputs, they produce the same outputs. Thus, to prevent attackers from being able to uncover/change secrets by replaying a victim's inputs, secure systems employ a random input (e.g., a key) to force attackers to guess one of the inputs. With sufficient randomness, the search space of the attacker is so large that the attacker's expected time to guessing correctly is longer than the Earth's lifespan, making an attack infeasible.

Broadly, modern cryptosystems employ two classes of random numbers: long-life keys that are pre-shared and ephemeral keys that are generated at run time. While both key classes are useful for providing confidentiality and integrity guarantees, pre-shared keys have an authentication component: having access to the key implies that you are a trusted entity; the fewer entities that have access to the same pre-shared key, the stronger notion of authentication provided. For symmetric cryptography, device-specific keys provide the strongest form of authentication, where only the device and its producer know the key. Historically, device keys were assigned pre-deployment and stored in a tamper-proof, non-volatile, memory inside the chip. Recent advances in harnessing analog-domain hardware chaos that results in non-deterministic cross-chip variation have replaced key storage with key generation via a Physical Unclonable Function (PUF). Ephemeral keys are generated using key agreement protocols, which rely on spontaneous generation of random numbers via a True Random Number Generator (TRNG). TRNGs also harness analog-domain chaos, but extract non-deterministic variation due to operational chaos to generate key material.

Internet-of-Things (IoT) devices are increasingly using PUFs and TRNGs. IoT devices are ubiquitous and often exposed to untrusted individuals, meaning they require the device identification provided by a PUF. At the same time, many medical, defense, safety-critical, and infrastructure systems make heavy use of IoT devices, meaning they are security-critical and require keys provided by a TRNG. But, by their nature, IoT devices must be small, cheap, and low power; such power and area constraints demand a unified hardware security primitive to fulfill both PUF and TRNG demands. An added benefit of a unified hardware security primitive is tamper evidence: validating PUF integrity also validates TRNG integrity.

Given the importance of PUFs and TRNGs to current IoT systems and the smart dust of the future, researchers attempt to provide a solution. The two most common unified hardware security primitives are based on Ring Oscillators (ROs) [33] and Static Random-Access Memory (SRAM) [21]. ROs send a signal around a ring of logic gates and using jitter as a source of chaos, while SRAM leverages a hardware-level race condition between two cross-coupled inverters

as a source of chaos. As, detailed in §2, there are many tradeoffs between RO- and SRAM-based PUFs and TRNGs, but in practice, SRAM-based hardware security primitives are more common [15, 22]. Unfortunately, SRAM's power-cycle-limited supply of entropy makes it ill-suited for use in a TRNG and its dual-use nature, while a way to reduce hardware overhead, is an attack vector [27, 30, 35, 47]. Thus, **there is no unified hardware security primitive capable of generating high rate and unbounded entropy, while minimizing hardware area and power, and is robust against attack**.

We observe that while it may seem that ROs and SRAM are completely different, they represent extreme design points in a continuum of hardware security primitive designs. We leverage this insight to create **RingRAM, a new unified hardware security primitive that combines the best aspects and avoids the drawbacks of both ROs and SRAM**. RingRAM exposes intermediate design points on the continuum, allowing system designers to balance PUF and TRNG utility.

RingRAM is composed of two cross-coupled, equal-length chains of an odd number of inverting gates (§3). When disabled, the two chains are effectively disconnected from each other. When enabled, the two chains race to send their value to the opposing chain; the chain that is relatively faster, determines the value for the cell. Relatively-different-speed chains are usable for PUFs, where relatively-similar-speed chains are usable for TRNGs (§8). The longer the chains, the more relatively similar the propagation delay through the chains, allowing a designer to control the PUF/TRNG composition of a set of RingRAM cells (§4).

To quantify the efficacy of RingRAM as a foundation for PUFs and TRNGs and to compare against RO- and SRAM-based primitives, we implement RingRAM on a Xilinx Artix-7 FPGA (§7). We develop three RingRAM implementations: FPGA, ASIC, and discrete. **We evaluate 1600 unique RingRAM cell sites, across 5 devices, over a 40° C range.** Experiments validate that RingRAM provides utility as the foundation for both PUFs and TRNGs and it combines the best aspects of RO- and SRAM-based primitives: **RingRAM provides unbounded entropy like ROs, has low hardware area cost like SRAM, has a higher throughput of entropy than either, and is more secure than either** (§8). To expose the middle of the PUF/TRNG continuum, we show how designers can extend RingRAM's design to create implementations that systematically target mid-points on the continuum (§4). We also show how small amounts of hardware can be added to reduce software's post-processing burden—clarifying the PUF and TRNG abstraction provided by hardware (§5) and to increase RingRAM's performance and security (§6). Lastly, we show how system designers can incorporate RingRAM into a System-on-Chip to improve overall system security without changing existing software or increasing software's run time (§10).

RingRAM makes the following contributions:

- RingRAM is a simple, auditable, high-performance, unified hardware security primitive for IoT devices (§3). We quantitatively validate RingRAM's suitability as the foundation for both PUF and TRNG uses as well as its overheads (§8).
- RingRAM exposes a tuning knob to designers that provides control over where in the continuum between PUF-oriented and TRNG-oriented a RingRAM implementation lies (§4).
- We expose knobs that reduce PUF noise and increase TRNG entropy rate (§5) as well as leverage device aging to increase both PUF and TRNG utility over time (§6).
- We implement RingRAM in a RISC-V-based SoC that runs Linux. Using this prototype, we demonstrate how RingRAM increases system security—without performance degradation—using openssl benchmarks (§10).

2 BACKGROUND

Physical Unclonable Functions (PUFs) and True Random Number Generators (TRNGs) are essential building blocks for the cryptographic systems that we all depend on. PUFs provide a means for strong device authentication and serve as a seed for key generation. TRNGs provide the ephemeral keys and nonces required to maintain data confidentiality and integrity. While security depends on PUFs and TRNGs, PUFs and TRNGs in-turn depend on non-determinism; note that this is different from a software notion of non-determinism, as even the most complex software is inherently deterministic. True non-determinism comes from chaos inherent to natural processes; in this case, variation in hardware's analog-domain. While hardware designers employ design practices and operational guard bands to mask analog-domain variation (e.g., voltage fluctuations) to create hardware that provides deterministic execution, PUFs and TRNGs must expose, enhance, and extract analog-domain variation.

The challenge is that not all variation is helpful for security. There are **two types of variation: systematic and chaotic**. Systematic variation is predictable, hence not useful, whereas chaos-induced variation is where the randomness lies. Thus, a hardware security primitive must eliminate systematic variation and capture chaotic variation. In addition, PUFs and TRNGs require mutually-exclusive types of variation. In this section, we provide an overview of PUFs and TRNGs, along with a description the most popular unified hardware security primitives that provide PUF and TRNG bits.

2.1 PUF

Strong device authentication (either post-deployment or while traversing the supply chain) and key generation for cryptographic protocols require a device-specific, on-chip key. This device-specific key must be robust against an attacker with physical access to the device, preventing them from exfiltrating it or duplicating it in another device. The traditional approach is to store the key in on-chip non-volatile memory (e.g., in Electrically-Erasable Programmable Read-Only Memory (EEPROM) or battery-backed Static Random-Access Memory (SRAM)) before deployment. Protecting against attackers with physical access requires tamper protection mechanisms, which increases complexity, cost, area, and power.

PUFs are a simple, low-cost, and naturally tamper resistant alternative to key storage: instead of assigning a key to a device, PUFs embody key derivation. **PUFs derive a device-specific key (i.e., a fingerprint) by harnessing manufacturing-time chaos that results in analog-domain hardware variation within and across chips**. The goal is to find sources of variation that result in reliable—but unpredictable—device differences that persist across its lifetime. Because the key depends on physical properties of the chip, it is naturally tamper-evident as physical modification changes the derived key. This obviates the need for the burdensome designed-in

tamper evidence required by key storage approaches. Thus, the **ideal primitive for a PUF enhances within-chip variation, diminishes wafer-scale variation, is tamper evident, and produces low-noise fingerprints that are stable with device use**.

2.2 TRNG

The security of the cryptographic protocols that society depends on rests on a small amount of non-predictability, conventionally called a key. Random Number Generators (RNGs) provide a stream of bits for use as key material. There are two types of RNGs, depending on the predictability of the produced bit stream: Pseudo-Random Number Generators (PRNGs) and True-Random Number Generators (TRNGs). PRNGs produce a sequence of bits that is **deterministically** derived from a seed value using a pseudorandom function (e.g., a cryptographic hash). Being calculable from a seed means that PRNGs have high throughput, but only the seed provides security; once the seed is known, all uses of the resulting PRNG output stream are compromised—including other keys/seeds derived from that sequence. In contrast, TRNGs produce a wholly **non-deterministic** sequence where every value is independent of previous values. Thus, every bit of the TRNG provides security.

While TRNGs are ideal from a security perspective, developers tend to avoid them due to their low throughput. **TRNGs achieve output independence by accumulating operational chaos**,[1] then distilling it down such that all possible N-bit output values have a probability of $1/2^N$ of occurring. Accumulating N-bits of chaos requires collecting much more than N-bits of analog-domain measurements, because only a small fraction of each measurement is chaotic—i.e., chaos is in the operational noise. The proportion of a measurement influenced by chaos is called entropy.[2] Thus, to create N-bits of true randomness, a TRNG must collect $entropy * N$-bits of measurements and reduce that down to an N-bit output.[3] Additionally, the measurement rate is also limited for many chaos sources, further reducing TRNG throughput.[4] The **ideal primitive for a TRNG maintains integrity, while providing an unbounded supply of entropy at a sufficient rate**.

2.3 Unified Hardware Security Primitive

Though there are many different types of PUF and TRNG designs, they tend to exploit similar hardware effects that are impacted by both manufacturing and operational chaos. Thus, the ideal solution of a single hardware security primitive that serves as the foundation for both PUFs and TRNGs is possible; we refer to this as a unified hardware security primitive. The two most popular unified hardware security primitives employ Ring Oscillators (ROs) [33] and Static Random-Access Memory (SRAM) [21]. At their lowest level, both ROs and SRAM use the basic inverter gate to sample chaos.

[1] Exposed sources of operational chaos are limited because they necessitate interaction with the environment, users, or require ground truth. The deeply-deployed nature of many IoT systems further eliminates sources of chaos.

[2] §8.2 covers entropy estimation algorithms in detail.

[3] A popular way to distil out the chaotic component of a largely deterministic set of measurements is to pass the accumulated measurements as the message to a compression function, taking the N-bit output as the TRNG response.

[4] For example, it takes AMD Ryzen processors 2500 clock cycles to provide 64-bits of true randomness [11].

2.3.1 *Ring Oscillators (ROs).*

ROs consist of an odd number of inverter gates connected together in a feedback loop. When active, it produces a value transition wave (i.e., 0 to 1 or 1 to 0 edge) that travels around the loop. The result is roughly equivalent to a clock signal with a 50% duty cycle, the frequency of which is dictated by the time it takes to circumnavigate the loop twice. Circumnavigation time is a combination of operational variation (e.g., voltage and temperature fluctuations) and manufacturing variation (e.g., threshold voltage). While manufacturing variation's effect on frequency is fixed for a given chip, operational variation consists of both systematic variation (e.g., 20°C vs. 50°C operation) and chaotic variation (e.g., thermal noise). As discussed in §8, ROs are better suited as a TRNG than a PUF due to their increased sensitivity to operational chaos; this sensitivity makes ROs vulnerable to environmental attacks.

2.3.2 *SRAM.*

At the heart of an SRAM cell is a pair of cross-coupled inverter gates. This cross-coupling creates a self-reinforcing bi-stable feedback loop that enables SRAM to maintain state without the need for the refresh operations of Dynamic Random-Access Memory (DRAM). The aspect of this cross-coupling that is useful for security comes when SRAM goes from un-powered to powered. When SRAM is un-powered, both inverters output 0, because they are off. When power is applied, the supply voltage rises quickly—but not instantaneously—due to power supply current limits and parasitic capacitance. Thus, during supply voltage rise, there is a point where one inverter is active (i.e., begins to output a 1), while the other remains off. The inverter that activates first is determined largely by manufacturing-time variation [46]. In cases where, due to random chance, inverters have very similar activation voltages, the inverter that wins the hardware race is, at least partly, determined by operational chaos.[5] Because of this, the power-on value of SRAM cells captures both manufacturing and operational chaos. As discussed in §8, SRAM is better suited as a PUF than a TRNG due to its increased sensitivity to manufacturing chaos; this sensitivity and their dual-use nature makes SRAM vulnerable to aging attacks.

We combine ROs and SRAM into a new unified hardware security primitive RingRAM. RingRAM has a designer-controlled balance between operational and manufacturing chaos that balances PUF and TRNG utility, while being immune to environmental and aging attacks.

3 DESIGN

While Ring Oscillator (RO) and Static Random-Access Memory (SRAM) security primitives seem different, we posit that they are fundamentally similar and representative of opposite extremes on a continuum of hardware security primitive designs. At the core of both ROs and SRAM is a simple combinational delay loop: ROs have long, self-inverting loops and SRAMs have short, self-reinforcing loops. ROs' long loops enhance operational variation at the expense of chaotic manufacturing variation, so ROs represent the True-Random Number Generator (TRNG) extreme of the continuum. SRAM's compact loops enhance chaotic manufacturing variation at the expense of operational and systematic manufacturing variation, so SRAM represents the Physical Unclonable Function (PUF) extreme of the continuum. The problem is that ROs and PUFs

[5] The compact and differential design of a SRAM cell naturally eliminates the effects of systematic operational variation.

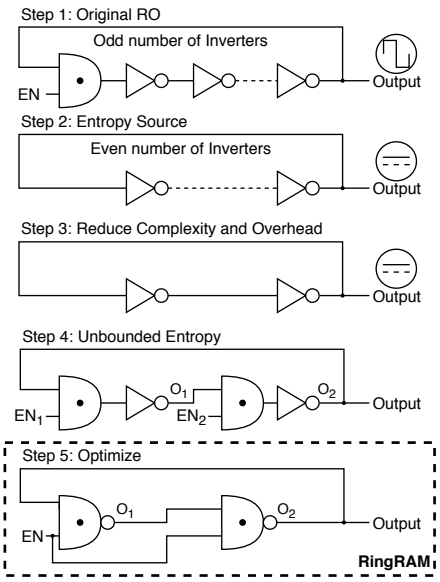

Figure 1: RingRAM design evolution.

are **extreme design points**, hence ill-suited for the unified hardware security primitive required by modern IoT systems. Even worse, both suffer critical security weaknesses.

We design a unified hardware security primitive RingRAM that is based on the same fundamentals as ROs and SRAM, but systematically designed from the ground-up to avoid the drawbacks of both ROs and SRAM. **RingRAM exposes and exploits the region of the hardware security primitive design space between the two extremes**. RingRAM combines the best design aspects of ROs and SRAM to create a single, simple, primitive that has low area overhead, provides an unbounded supply of high-rate entropy, provides a robust device fingerprint, while addressing their security weaknesses. While we develop RingRAM from the ground-up, based on first principles, for clarity here, we describe its construction starting from a RO and iteratively modify the design to eliminate negatives. Figure 1 shows RingRAM's evolution.

Step 1 - Base RO: ROs provide an unbounded amount of chaos-influenced values, controlled by an *AND* gate. Unfortunately, because ROs are sensitive to both systematic and chaotic operational variation, the rate of entropy is low as the TRNG must wait for sufficient chaotic operational variation to accumulate such that it surpasses systematic variation. Additionally, the long chains of ROs tend to average out chaos-induced manufacturing variation.

Step 2 - Entropy Source: In comparison to ROs, the entropy source of SRAM is based on stabilization as opposed to noise accumulation. The hardware-level race condition created by SRAM's cross-coupled inverters is sensitive to both manufacturing and operational chaos, albeit much more sensitive to manufacturing chaos. By switching to SRAM's entropy source, there is no need to wait for noise accumulation, increasing throughput, and because it is sensitive to both sources of chaotic variation, area overhead decreases because only one ring is required per PUF response bit.

Previous State		Current State		O_1	O_2	Comment
EN_1	EN_2	EN_1	EN_2			
-	-	**0**	**0**	1	1	**No Feeback**
-	-	0	1	1	$\overline{O_1}$	Stable 1
-	-	1	0	$\overline{O_2}$	1	Stable 0
0	**0**	**1**	**1**	$\overline{O_2}$	$\overline{O_1}$	**Race Condition**
0	1	1	1	1	$\overline{O_1}$	Stable 1
1	0	1	1	$\overline{O_2}$	1	Stable 0
1	1	1	1	O_1	O_2	Hold output

Table 1: RingRAM logical operation

Step 3 - Reduce Complexity and Overhead: An even number of inverters in a feedback loop produces a stable response that is dictated by the inverter that first drives the others at power-on: a race condition. However, a race condition requires only two inverter gates—there is no longer a need for the 100+ inverters required by a RO. Thus, to minimize area overhead and complexity of our primitive, we reduce the feedback loop to two inverters. Eliminating long inverter loops reduces area overhead, but biases the sensitivity of our primitive toward manufacturing chaos and away from operational chaos, due to the loss of the averaging effect. In §4, we systematically increase the number of inverters in a loop to show how hardware designers can tune RingRAM's bias—choosing a point along the hardware security primitive design space.

Step 4 - Unbounded Entropy Even though the result of step three is a more efficient structure, it is inherently bounded by requiring a power cycle to produce a new response. ROs produce unbounded responses by splicing an *AND* gate into their feedback path; this acts an an enable. While it may seem that this is a different effect than a power cycle, we observe the practical effect of a power cycle is to set all wires in the loop to 0. By splicing an *AND* gate into the start of each leg of the cross-coupled feedback path (i.e., before each inverter input in this case) the wires in the loop will be forced to 0—without a power cycle. Table 1 provides a truth table of our primitives behavior. The key is that going from disabled to enabled activates a race condition between the inverters, like a power cycle in SRAM—yielding unbounded entropy.

Step 5 - Optimize Implementing *AND* gates and inverters requires the use of redundant inverters, as a *AND* gate consists of a *NAND* gate and an inverter at the transistor level. By replacing each *AND* and inverter pair with a *NAND* gate, we reduce the required number of transistors from 16 to 8 per cell. We further minimize complexity by utilizing a single enable signal for both *NAND* gates, as this does not change the premise of the race condition. The final result of our design is a simple unified hardware security primitive that provides unbounded entropy at a higher rate than ROs, is nearly as small and compact as SRAM, and is resistant to known attacks.

4 CONTROLLING COMPOSITION

Manufacturing variation tends to produce inverters with different turn-on voltages for a given SRAM cell. This means that the base design of RingRAM heavily favors producing cells where one inverter reliably wins the hardware race over the other. While this is great for PUFs, it severely limits the entropy provided by a set of RingRAM

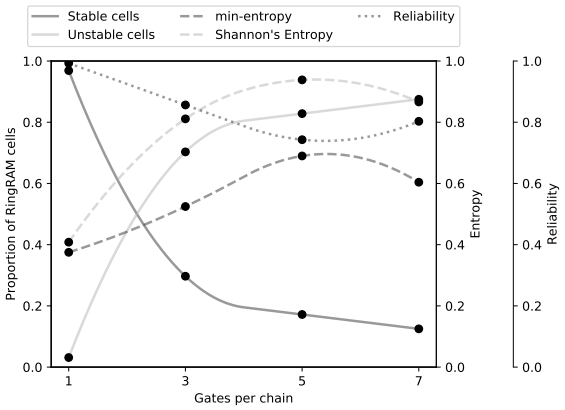

Figure 2: Increasing the number of gates per cross-coupled chain increases the proportion of unstable cells usable by TRNGs and the resulting entropy.

cells, which require more random race outcomes. A simple solution is to add more cells, thereby increasing the entropy—at least probabilistically. Unfortunately, this simple solution is inadequate, because it requires adding roughly 32 RingRAM cells for every unstable cell required to meet TRNG entropy requirements.

A superior solution is to expose a knob that designers use to modulate the proportion of unstable cells for a given set of cells. This way, designers can tune the composition of cells to match their specific needs, without the hardware overhead of the naïve solution. Unstable cells are those with relatively similar gate delays; hence, the goal is to create chains (i.e., half of the cross-coupled feedback loop) with similar propagation delays. To accomplish this, we leverage the law of large numbers [12]: "the average of the results obtained from a large number of trials should be close to the expected value and will tend to become closer to the expected value as more trials are performed." In RingRAM's case, you can view each gate in a chain as a trial from a distribution of manufacturing-time variation; longer chains equate to more trials, hence more uniform total delay. Thus, as chain length increases,[6] it becomes more likely that a cell has chains with similar propagation delays, i.e., is unstable.

To evaluate our ability to control RingRAM composition, we implement 64 RingRAM cells on the FPGA platform from §8 and collect 320K responses from each cell, repeating this for 2, 4, and 6 additional gates. We augment RingRAM's layout to ensure longer chains are strict extensions of shorter chains. The results in Figure 2 show that increasing the number of gates per chain averages-out manufacturing-time variation, increasing the proportion of unstable cells. This, in turn, increases both min-entropy and Shannon's Entropy (see §8.2), with diminishing returns. We observe that this is the result of more slightly unstable cells being added than perfectly (i.e., 50%) unstable cells. The net effect is **a reduction in the number of transistors required to produce an unstable cell by between 44% and 60%**, compared to adding more cells. Thus, this is an effective approach to control where a RingRAM implementation lies on the PUF/TRNG continuum.

[6]Note that each chain must have an odd number of gates to produce a bi-stable circuit when cross-coupled with the opposing chain.

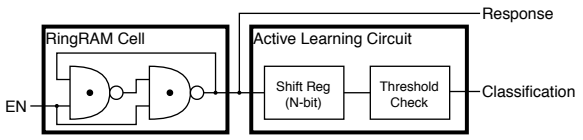

Figure 3: The active learning circuit dynamically classifies cells as either stable or unstable based on a brief, recent, history of cell responses.

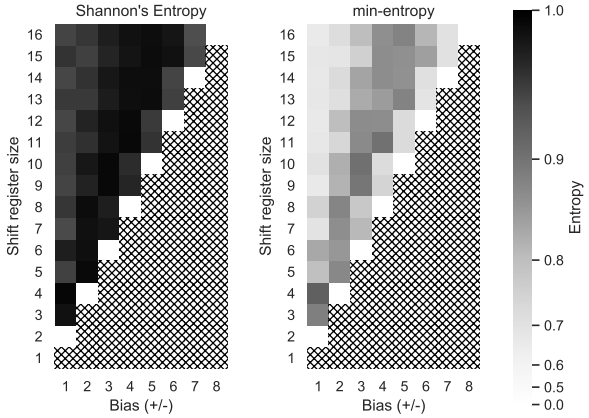

Figure 4: Active learning increases entropy in the TRNG response across configurations, with a 4-bit shift register ±1-bit bias being optimal in terms of entropy and hardware overhead.

5 ON-CHIP PROCESSING

While increasing the number of gates per chain does increase the proportion of unstable cells (i.e., total entropy), it does not alleviate the burden on software to distill-out randomness (i.e., entropy throughput). As explained in §8.2, software must take many more RingRAM responses than it needs bits-of-randomness, because each RingRAM output is only partially influenced by chaos. In the case of the base RingRAM design on our FPGA, this influence ranges between .02 and .06 random-bits per RingRAM response bit, depending on entropy metric. Even with controlled composition, entropy is still below 1.0. A second source of concern is the noise in the PUF response, which increases as chains lengthen. Thus, the current RingRAM interface still requires extreme software post-processing to separate and refine concerns.

The root cause of these issues is the mixing of PUF response and TRNG response. The **ideal unified hardware security primitive provides separate PUF and TRNG interfaces, where the PUF response is low noise and the TRNG response is high entropy**. To achieve this ideal, we enhance RingRAM's design with the ability to dynamically partition cells into stable and unstable and send the cell responses to their respective, software accessible, interfaces. This increases the entropy throughput to software, but the total entropy provided by a set of RingRAM cells stays the same.

Active learning classifies each RingRAM cell by first collecting several of its responses. If the number of 1's in the response is within a threshold (determined experimentally later) of the 50% count, then the cell is marked at unstable; otherwise, the cell is marked as stable. The TRNG register collects only unstable cell responses, increasing throughput to software, while the PUF register replaces unstable cell

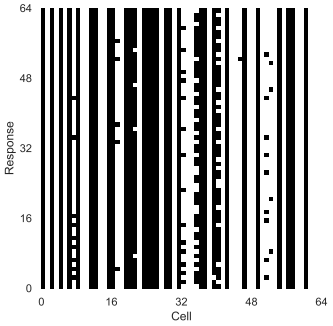

Figure 5: **RingRAM's single-interface responses.**

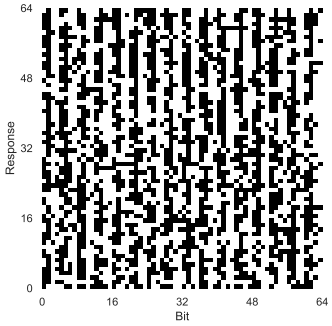

Figure 6: **Statically-classified RingRAM responses using enrollment data.**

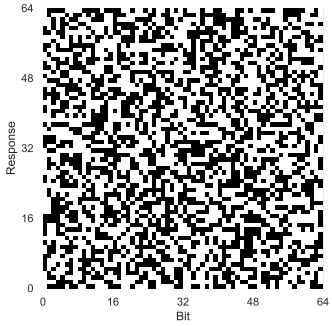

Figure 7: **Active learning's dynamic classification response with a 4-bit shift register and ±1 bias.**

responses based on bias, reducing noise. Figure 3 depicts how the active learning circuit integrates with a RingRAM cell.

This design presents two knobs to designers: shift register size and classification threshold. Shift register size dictates how many bits are available for the classification decision. Increasing shift register size reduces misclassification at the cost of increased area. On the other hand, the classification threshold (i.e., bias) dictates the allowable number of bits opposing the majority for a cell to be classified as unstable. A high bias, relative to shift register size, produces more unstable cells, increasing total entropy and TRNG throughput, while reducing PUF noise. A low bias results in more stable cells, increasing average entropy and the number of PUF bits.

To determine the optimal parameter settings, we sweep through both parameters with an emphasis on entropy (because RingRAM is PUF-dominant). For this exploration, we use the 5-gate-per-chain 64-cell FPGA implementation from §4. For each parameter combination, we capture 320K TRNG responses. Figure 4 shows the results for both entropy metrics in a two-dimensional heat map. In all cases, active learning increases entropy compared to the original design (the white boxes on the upward diagonal). We observe that, across shift register sizes, the optimal bias is ±1; even cells with low entropy are helpful to the overall TRNG performance. **A 4-bit shift register maximizes entropy, while minimizing hardware overhead**. The resulting 4-bit±1-bit configuration yields a Shannon's Entropy >0.9999 **(+16x)** and a min-entropy of 0.981 **(+35x)**, meaning software has to waste up to 35x less time acquiring and processing RingRAM data.

Besides reducing software's burden, it turns out that active learning's dynamic classification has another advantage: **dynamic classification better leverages bursts of randomness**, while avoiding bursts of uniformity than static classification.[7] The progression of TRNG responses from the base, single interface design shown in Figure 5, to software classification shown in Figure 6, to the active learning's output shown in Figure 7 makes clear the advantage of dynamic, on-chip classification.

6 GETTING BETTER WITH AGE

Unlike software, electronic devices change over time due to operational and utilization effects in a process called aging. There are four main contributors to device aging: Hot-Carrier Injection (HCI), Time-Dependant Dielectric Breakdown (TDDB), Electro-Migration (EM), and Negative-Bias Temperature Instability (NBTI) [27]; the most significant for modern transistors being NBTI [34]. NBTI increases a gate's threshold voltage (roughly, the voltage that it requires at its input to start outputting a logic 1). This makes it slower to switch from a 0 to a 1. NBTI is due to the accumulation of contamination in the dielectric of a transistor (specifically a PMOS) [23].[8] Because contamination only occurs when a transistor is conducting charge between the source and the drain (i.e., on), NBTI in cross-coupled stable circuits like RingRAM and SRAM is data dependent.[9]

While data-dependent aging is a serious security risk for SRAM-based hardware security primitives [27, 30, 32, 35], **we leverage aging to improve RingRAM's performance** for both PUFs and TRNGs. We observe that we can direct device aging to gradually influence the relative speeds of the cross-coupled chains: making stable cells more stable and unstable cells more unstable. To harness device aging to improve RingRAM's performance, we intentionally have cells hold a value while idle. For stable cells, we have them hold a value that makes them more stable over time. For unstable cells, we have them hold a value that makes them more unstable over time. To determine the value, we observe that the faster chain dictates the cell's value. Hence, to make a cell more stable, we invert the value, loading a value that causes the loser to turn-on and commence aging. By extension, for unstable cells, we let the winner keep aging, making the race closer next time.

To implement this functionality, we leverage the active learning circuit. To this we add an aging mode that loads values to each RingRAM chain dependent on cell type and response history. Figure 8 shows the directed aging circuit and Table 2 provides a logical description. When RingRAM is idle, the directed aging circuit forces the cell to hold values based in its recent value history.

Evaluating the exact impact of the directed aging circuit on RingRAM's performance is infeasible as it requires running the FPGA prototype for many years. Short of that, we approximate the effects of aging by leveraging the similarity in the core structure

[7] By dynamically classifying bits as random, RingRAM with active learning adds a new spatial dimension on top of per-cell randomness. This reordering increases entropy for free.

[8] Positive-Bias Temperature Instability is also a source of device aging, but has been shown to be less significant than NBTI. While both forms of BTI affect RingRAM, we focus on NBTI because it dominates and for simplicity.

[9] NBTI affects ROs as well: it causes them to slow over time. This results in reduced entropy rate for RO-TRNGs. Also, given asymmetries in susceptability to aging between transistors, this adds noise to RO-PUFs.

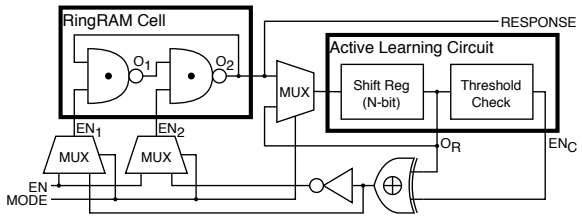

Figure 8: **The directed aging circuit gradually reduces PUF noise, while increasing TRNG entropy, by aging cells according to values in the active learning circuit's shift register. Stable cells are set to values inverted from what is in the register, further slowing the already slow gate. Unstable cells are set to values matching those in the register, slowing the faster gate.**

MODE	EN	EN_C	EN_1	EN_2	O_1	O_2
0	0	-	0	0	0	0
0	1	-	1	1	$\overline{O_2}$	$\overline{O_1}$
1	-	0	O_R	$\overline{O_R}$	$\overline{O_R}$	O_R
1	-	1	$\overline{O_R}$	O_R	O_R	$\overline{O_R}$

Table 2: **Directed aging circuit's logical behavior. `MODE` is 1 when directed aging is enabled, EN_C encodes whether the cell is a stable/PUF cell (1) or a unstable/TRNG cell (0), and O_R is the output from the cell's shift register. O_1 and O_2 are the individual chain output values, regardless of chain length.**

of SRAM and RingRAM. We gather 5 years worth of aging data using four Texas Instruments MSP430G2553 launchpad development boards as testbeds. We first enroll the boards by taking 51 measurements of their power-on state. Since NBTI aging is data-dependent, we write all `1`'s to half the boards and all `0`'s to the other half. After 5 years of accelerated aging [27], we take another 51 power-on state measurements and compare to the enrollment data. The results of down-selecting bits that match what directed aging would have done are (1) **PUF response noise reduces to ±2-bits (-40%)** and (2) **min-entropy increases by 122%**. This makes sense as directed aging aggressively moves unstable cells to 50% probability (±7%), which is what min-entropy is sensitive to. There is a synergy between directed aging and **controlled composition: controlled composition creates partially-unstable cells and directed aging gradually makes them less stable**.

7 IMPLEMENTATION

RingRAM relies on the idea of a self-reinforcing, bi-stable feedback loop that can be programmatically forced to an unstable point and will eventually stabilize itself to a value that depends on both manufacturing and operational chaos. While there are a myriad of ways to implement such functionality, we provide an area-optimal implementation comprised of 2 `NAND` gates (§3). This construction creates a hardware-level race condition when enabled (Table 1). Using such simple components, RingRAM is implementable using discrete circuit components, Field Programmable Gate Arrays (FPGAs), and as part of an Application-Specific Integrated Circuit (ASIC).[10]

There are **two sources of variation that a designer must avoid when implementing RingRAM: systematic and structural**. As discussed in §2, systematic variation occurs both at manufacturing- and run-time. Systematic variation occurs due to predictable changes in transistor properties at chip- and wafer-scale and due to long-running changes in a device's operational environment. Alternatively, structural variation is a universal difference in the placement and routing of RingRAM components. For example, an enable register being closer to one `NAND` gate than the other means the electrical signal will reach the closer gate first, making it more likely to win the hardware race. Both types of variation distort the PUF/TRNG-cell ratio of RingRAM towards a PUF, while making the resulting PUF responses more predictable across devices. We provide two guidelines to avoid these sources of variation:

(1) **Symmetric:** As both components and wires have the potential to add structural variation, use symmetrical placement and routing to avoid structural variation.
(2) **Tightly-packed:** By keeping the chains of a cell physically adjacent and its routing short, there is little room for systematic variation to influence chains asymmetrically.

7.1 Discrete Implementation

Implementing RingRAM using discrete components affords hardware designers full control over RingRAM's composition and the resulting PUF response by hand-tuning the layout and parasitics. Obviously, for threat models where the attacker has physical access to the device, additional physical anti-tamper measures are necessary to protect the primitive's integrity [41]. We implement RingRAM using four discrete Bipolar Junction Transistors (BJTs) as Figure 9 shows. We follow our implementation guidelines and layout the BJTs symmetrically and pack them tightly. This implementation creates the same race-condition between two `NAND` gates, where the `NAND` gates are created using two `NPN` BJTs, connected in series. When the base (i.e., center pin) of the `NPN` transistor is 1, it acts as a short-circuit between the other two pins. Therefore, only when both `NPN` BJTs that are connected in series have their base set to 1 is the output of the pair 0. This construction preserves the required hardware-level race-condition as `O1` and `O2` contend with each other until the cell stabilizes. In this setup, whether the cell is useful for a PUF or TRNG depends on the relative properties of the transistors and the resistors. Thus, tuning the bias of the cell is possible using variable resistors.

7.2 HDL Implementation

Implementing RingRAM in a Hardware Descriptive Language (HDL) presents challenges due to it's required combinational feedback loop, see Listing 1. By default, HDL synthesis and implementation tools prevent the use of such feedback loops as they run contrary to synchronous design practices. Also, RingRAM's self-reinforcing loops appear to be redundant at the digital abstraction level, so we must prevent the tools from optimizing them away. To solve this challenge we leverage three flags:

- **ALLOW_COMBINATORIAL_LOOPS:** prevents feedback loops from being flagged as errors
- **KEEP:** prevents the component/wire from being removed
- **DONT_TOUCH:** prevents the component/wire from being optimized away.

[10] An important aspect of the of the discrete and FPGA implementations is that they are auditable and able to be integrated with existing systems. Auditability is important given recent distrust in black-box TRNGs [17].

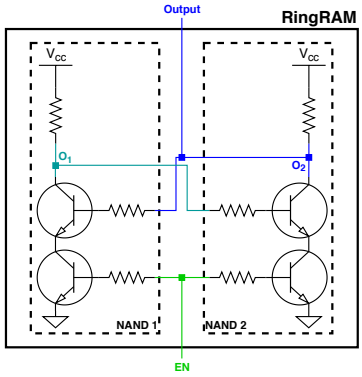

Figure 9: RingRAM NPN BJT schematic.

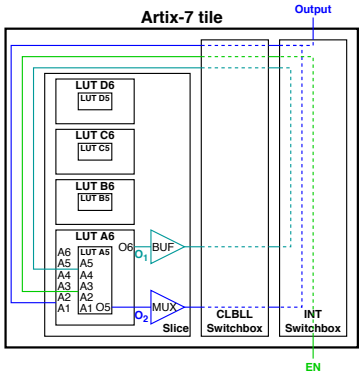

Figure 10: RingRAM FPGA LUT schematic.

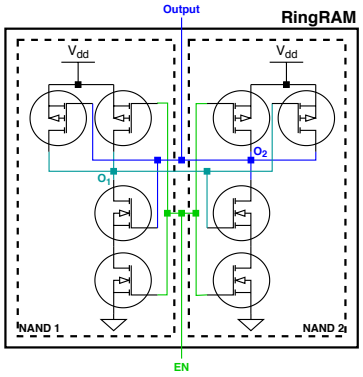

Figure 11: RingRAM CMOS ASIC schematic.

```
module evalRingRAM(
    input  clk,
    input  en_in,
    output reg out);

    (* ALLOW_COMBINATORIAL_LOOPS = "true",
       KEEP = "true",
       DONT_TOUCH="true" *)
    wire nand_1, nand_2;
    reg en;

    assign nand_1 = !(en & nand_2);
    assign nand_2 = !(en & nand_1);

    always @(posedge clk) begin
        en <= en_in;
        out <= nand_2;
    end
endmodule
```

Listing 1: RingRAM hardware description used for FPGA and ASIC.

7.2.1 FPGA. FPGAs contain arrays of programmable logic blocks and configurable interconnects, laid out in a regular pattern, which allow designers to implement circuits rapidly and at low cost. Logic gates, e.g., NAND gates, are implemented using Look-Up Tables (LUTs). LUTs encode an input to output function that emulates the behavior of the circuit they are implementing. FPGA logic blocks are categorized into slices, where each slice contains (in our FPGA): 4 logic-function generators (i.e., LUTs) with 6 inputs to 2 outputs, 8 storage elements, wide-function multiplexers, and carry logic [43]. As a single RingRAM cell requires two NAND gates, we implement up to four RingRAM cells per slice.

The major challenge with implementing RingRAM on an FPGA is maintaining symmetry given the fixed and closed-source nature of FPGA physical layouts. We bypass the auto-router by forcing the placement of each RingRAM cell and the ports used on the LUT to connect the components together. To meet our tightly-packed requirement, we use vertically stacked LUTs for the NAND gates of a RingRAM cell, because vertically-stacked slices are directly connected via fixed wires. To minimize skewing the delay on our enable wires (i.e., meet our symmetry requirement), we place an enable buffer either in proximity to two symmetrical LUTs or in an adjacent slice. Figure 10 shows the results of our automated RingRAM placement and routing tool on a Xilinx Artix-7 FPGA.

	Utilization			Timing	
	LUT	LUTRAM	FF	WNS	Max Freq
RingRAM(1)	1	0	2	8.315ns	593MHz
RingRAM(3)	6	0	2	7.609ns	418MHz
RingRAM(5)	10	0	2	6.527ns	288MHz
RingRAM(7)	14	0	2	5.197ns	208MHz
RingRAM(5)+Active Learning	18	1	8	5.713ns	233MHz
RingRAM(5)+Active Learning+Active Aging	19	1	9	5.449ns	220MHz

Table 3: Artix 7 overhead and performance

Table 3 show the hardware area and timing cost of the various RingRAM design-space options.

7.2.2 ASIC. An ASIC implementation of RingRAM engenders a more symmetrical and compact design than is possible in FPGA and discrete implementations. This is possible because ASIC implementations are not restricted by the fixed placement and routing options provided by FPGAs. Area overhead is also reduced compared to FPGA implementations, because there are no wasted transistors: our FPGA implementation wastes the majority of a LUT's capability. As Figure 11 illustrates, our ASIC RingRAM implementation requires eight transistors (four per NAND gate). This provides near-SRAM-levels of density, as traditional SRAM cells consist of six transistors (two per inverter and two access transistors). Taken together, ASIC RingRAM implementations, on average, outperform FPGA implementations for both PUF and TRNG, due to the reduced systematic variation and structural bias.

8 EVALUATION

A unified hardware security primitive must serve as the foundation for both Physical Unclonable Functions (PUFs) and True Random Number Generators (TRNGs). In earlier sections, we design and implement RingRAM for this purpose. RingRAM is a hardware security primitive that leverages a simple, bi-stable, feedback loop that captures both manufacturing and operational chaos. In this section, we discover where RingRAM resides on the PUF/TRNG continuum, then validate that it is a unified hardware security primitive.

[11] RO PUFs achieve high reliability by sacrificing area overhead and latency, averaging up to 4 billion comparisons. A 64-bit RO-PUF requires 128 ROs (21.8%), two 32-bit counters (25.6%), a 32-bit subtractor (15.0%), two 1:64 DEMUXs (18.8%), and two 64:1 MUXs (18.8%) [28].

	Metric	RO [28, 36]	SRAM [1, 21]	RingRAM
Security	Single-use	✓	× [30, 47]	✓
	Aging Resilient	✓	× [27, 30, 35]	✓
	Thermal Resilient	× [4, 40]	✓	✓
	Voltage Resilient	× [3]	✓	✓
PUF	Reliability	99.1%[11]	92.2%	98.4%–94.4%
	Uniformity	49.4%	48.7%	48.2%–47.2%
	Uniqueness	47.2%	48.7%	48.4%–49.9%
TRNG	Unbounded	✓	×	✓
	Throughput	38M	N/A	210M–228M
	min-entropy	0.97	0.031	0.351–0.981
	Shannon's Entropy	0.99	0.058	0.423–0.999
	Transistors/unified bit	1641.05	99.75	7.5–88.5

Table 4: RingRAM combines the best aspects of RO- and SRAM-based hardware security primitives—without their security weaknesses. The range presented for RingRAM represents the base design on the left to the 5-gate active learning on the right.

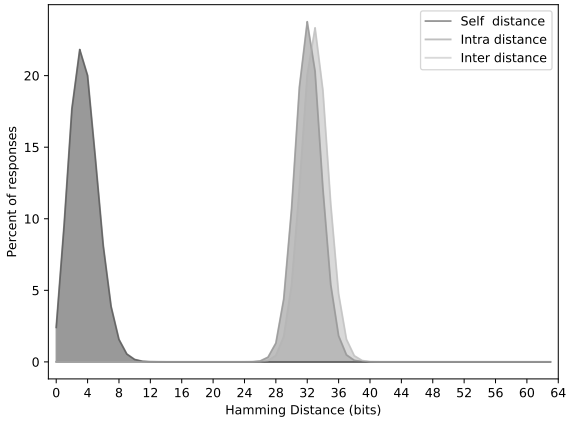

Figure 12: Distance in 64-bit responses from the same location on the same FPGA (`Self`), from the same location on four other FPGAs (`Inter`), and from different locations on the same FPGA (`Intra`). RingRAM's responses are near the ideal of 0-bits for `Self` and 32-bits for `Inter` and `Intra`.

The base system for all experiments is a 5-gate-per-chain version RingRAM with active learning, implemented on the Xilinx Artix-7 FGPA [45] on a Digilent Arty-A7-100T evaluation board [13].[12] Inside the FPGA, we implement a set of 64 RingRAM cells that are broken-out dynamically into a 64-bit PUF response and a 64-bit TRNG output by the active learning circuit. A custom state machine reports the PUF response and TRNG output to a data collection computer via a UART controller. A program running on a desktop computer saves the UART data as a binary file.

8.1 RingRAM PUF

RingRAM provides ample stable cells, but are they useful for a PUF? For use in a PUF, stable cells must repeatedly produce the same response (i.e., reliable), while being non-biased (i.e., uniform), and be device dependent (i.e., unique). To asses utility in a PUF, we evaluate sets of 64 cells to prove that RingRAM meets these criteria, quantitatively comparing to RO- and SRAM-based PUFs.

Reliability: The reliability of a cell is quantified by its ability to repeatedly produce the same responses. To quantify response variance, we take 320K responses from our 64 cells, randomly select one measurement to be the reference fingerprint, and calculate the Hamming Distance between each measurement and the reference. Figure 12 shows a histogram of these results as `Self distance`. The graph follows a Gaussian distribution centered on 4 bits, indicating that the resulting fingerprint contains 56 bits of potential device-discrimination ability. To quantify reliability in a scalar value, we evaluate response variation using Equation 1 [28], where m is 320K responses, n is 64-bits, R_0 is the enrolled response, and R_i is from the set of responses to compare to.

$$Reliability = 100\% - \frac{1}{m}\sum_{i=1}^{m}\frac{HD(R_0, R_i)}{n} \times 100\% \quad (1)$$

RingRAM's measured reliability is 94.5%, as shown in Table 4, a 2.5% increase over SRAM and a 4.8% decrease from a RO. This is expected as SRAM's more compact and controlled layout yields a higher rate of TRNG cells than an FPGA implementation, thus has to contend with more noise in its fingerprint. We expect an ASIC RingRAM implementation to be closer to SRAM.

Uniformity: The uniformity of a set of cells is their ability to produce balanced responses. A perfectly uniform response is a 1:1 ratio of 0's and 1's, otherwise there exists predictability that reduces utility. Like SRAM, RingRAM responses are uniformly distributed when looking at many cells. To quantify uniformity in a scalar value, we determine the ratio of 1's and 0's across the set of 320K responses from just 64 cells using Equation 2 [28], where the variables match those in Eq. 1 and HW() is the Hamming Weight.

$$Uniformity = \frac{1}{m}\sum_{i=1}^{m}\frac{HW(R_i)}{n} \times 100\% \quad (2)$$

As shown in Table 4, RingRAM's uniformity of 47% is on-par with ROs and SRAM. We expect ASIC implementations to have a small increase in uniformity due to tighter controls over cell layout that increases symmetry.

Uniqueness: Uniqueness is dictated by a response's dependency on placement both within a chip (intra-chip distance) and across the same location on multiple chips (inter-chip distance). We determine intra-chip distance by measuring the Hamming Distance between 5 sets of 64 RingRAM cells placed in different locations in a single chip and sampled 320K times. We determine inter-chip distance by measuring the Hamming Distance between 5 sets of 64 RingRAM cells placed in the same location on five different chips and sampled 320K times. Figure 12 shows that both intra- and inter-chip distances are similar: 32-bits with a 99.9% confidence interval of +/- 3 bits. Therefore, RingRAM's responses have a 29 bit worst-case difference, which is close to the ideal of 32-bits. More importantly, there is a vast gulf between the distances between responses from the same location and chip than from either other locations or other chips.

To quantify uniqueness, we condense the intra- and inter-chip response differences using Equation 3 [28], the variables match those

[12] While we select an FPGA implementation for cost and speed reasons, we expect the evaluation trends to hold for both discrete and ASIC implementations. Because of their nature, we expect discrete implementations to be slightly more PUF oriented (due to increased systematic and structural variation) and ASIC implementations to be slightly more TRNG oriented (due to compactness and symmetry) than the FPGA.

Statistical Test	P-Value	Result
Monobit	0.839618578844032	PASS
Frequency Within a Block	0.0611787903895858	PASS
Runs	0.25424654016127324	PASS
Longest-Run-of-Ones in a Block	0.4602661335812786	PASS
Binary Matrix Rank	0.060600309853307055	PASS
Discrete Fourier Transform	0.831370987522874	PASS
Non-Overlapping Template Matching	0.9999962973281614	PASS
Overlapping Template Matching	0.490469422296025	PASS
Maurer's "Universal Statistical"	0.9989051372547096	PASS
Linear Complexity	0.4876367178175764	PASS
Serial	0.033925539965049746	PASS
Approximate Entropy	0.05939790977947617	PASS
Cumulative Sums	0.8750549175582036	PASS
Random Excursions	0.024767950538244106	PASS
Random Excursions Variant	0.1453894509643879	PASS

Table 5: RingRAM passes the NIST test suite.

in Eq. 1, except R_i and R_j are responses from chips/locations i and j, respectively, and c is the number of chips/locations.

$$Uniqueness = \frac{2}{c(c-1)} \sum_{i=1}^{c-1} \sum_{j=i+1}^{c} \frac{HD(R_i, R_j)}{n} \times 100\% \quad (3)$$

RingRAM's measured intra-chip uniqueness is 46% and inter-chip uniqueness 50%, giving RingRAM an average uniqueness of 48%. RingRAM is both device- and position-dependent, producing uniqueness similar to RO and SRAM PUFs (Table 4).

8.2 RingRAM TRNG

For RingRAM to be a universal hardware primitive we must also validate that its unstable cells are usable by TRNGs. Unstable cells are created when the cross-coupled gates activate at nearly the same voltage (i.e., time after enable goes high). The design of RingRAM assumes that in such cases, chaotic operational variation controls the winner of the hardware race condition. When this occurs, the resulting cell response encodes some amount of chaos. In a TRNG context, the amount of influence chaos has over a cell's response is referred to as its entropy; 0 entropy means the response is totally deterministic, while 1.0 entropy means the response is totally non-deterministic (i.e., completely random).

The goal of this experiment is to determine RingRAM's entropy, which dictates how many responses are required to produce a truly random N-bit key. Unfortunately, this is a known challenging problem that can only be approximated with entropy estimation metrics. Though there are many different entropy metrics, the two most popular metrics are min-entropy and Shannon's Entropy.[13] min-entropy is a worst-case metric that quantifies the guessing odds of an attacker that has a large number of previous RingRAM responses and uses that information to increase their chances of guessing the next response correctly. As shown in Equation 4, the best strategy is for the attacker to guess the most likely response.

$$min\text{-}entropy = log_2 \frac{1}{P_{MAX}(x)} \quad (4)$$

[13]These two metrics come from the Rényi family of entropies, representing the worst and average case, respectively [6].

The result is the number of bits of randomness provided by the number of bits in X (64-bits in RingRAM's case). To determine entropy, which is randomness-per-response-bit, divide `min-entropy` by the number of bits in X.

Most threat models do not afford such a powerful attacker, hence we look at Shannon's Entropy. Shannon's Entropy is an average-case metric that quantifies the guessing odds of an attacker under the assumption that they have no special understanding of the system. As shown in Equation 5, there is no better strategy than blind guessing; so the metric considers how far every possible response diverges from the expected even probability (i.e., $\frac{1}{2^{|X|}}$).

$$Shannon\ Entropy = -\sum_{i=0}^{n} P(x_i) log_2 P(x_i) \quad (5)$$

The result is treated the same way as that of `min-entropy`.

To calculate RingRAM's entropy we collect 320K responses from each of 64 cells. Because collecting enough responses to see sufficient duplicates in a space of 2^{64} possible responses is infeasible, we leverage the earlier observation of cell independence[14] to break the problem into a combination of 8, 8-bit responses. For each 8-bit chunk, we track the frequency every possible response. We create an entropy for the entire 64-bits by averaging the 8, 8-bit entropies. Table 4 shows that RingRAM's min-entropy is 0.981 and Shannon's Entropy is >0.9999, better than SRAM- and RO-based TRNGs. RingRAM also provides increased entropy throughput (i.e., random-bits-per-second) on our FPGA. **Given RingRAM's high throughput advantage and unbounded nature, it represents the superior primitive for a TRNG**.

Entropy only quantifies predictability, it does not account for statistical patterns in the responses. The National Institute of Standards and Technology (NIST) provides a statistical test suite used to validate the entropy of TRNGs. While five-gate chains combined with active learning provides high levels of Shannon's Entropy and min-entropy, those are coarse-grain metrics used to quantify the potential for randomness. We use the NIST test suite to perform a more comprehensive set of statistical tests using a SHA256-based distilation of RingRAM TRNG outputs to meet the input requirements of the NIST test. We compress $\lceil \frac{256}{.98} \rceil$-bits of RingRAM responses down to 256-bit values and pass them to the NIST test. As the results in Table 5 show, RingRAM passes the entire NIST test suite.

9 RINGRAM'S THERMAL STABILITY

As mentioned in §2, a significant source of systematic run-time variation is the temperature. Temperature has a known deleterious effect on the performance of ROs due to their inability to filter-out systematic run-time variation (especially for RO TRNGs). To eliminate uncontrolled thermal deviations from contaminating our results in earlier experiments, we used a Test Equity temperature and humidity chamber (Model 123H) to fix the temperature 20°C.

While the cross-coupled nature of RingRAM should filter-out systematic run-time variation, making it robust to temperature and voltage changes, we experimentally verify this assumption. To do so,

[14]We select 8-bit symbol sizes so that our results directly compare to previous work [21]. Our script reports results from 1- to 16-bits, but every additional symbol bit requires double the number of responses to remove the impact of random variations from the resulting entropy.

	Temperature	0°C	10°C	**20°C**	30°C	40°C
PUF	Self Distance	-0.5089	-0.0811	**3.5712**	-0.4702	-0.8141
	Reliability	+0.7952%	+0.1268%	**94.4199%**	+0.7347%	+1.2721%
	Uniformity	-0.6324%	-0.1589%	**47.1771%**	+0.4278%	+0.7836%
	Uniqueness-Intra	-1.0417%	+0.0001%	**45.8333%**	+0.0001%	-1.0417%
	Uniqueness-Inter	+2.0833%	+0.0001%	**49.9999%**	+3.1250%	+3.1250%
TRNG	min-entropy	+0.0022	-0.0019	**0.981**	-0.0007	-0.0008
	Shannon's Entropy	+0.0001	-0.0001	**>0.9999**	-0.0001	-0.0001

Table 6: RingRAM is robust against thermal variation.

we re-run earlier experiments, except this time, we vary temperature between 0°C and 50°C, in 10°C increments. For the PUF validation experiments, we use the original 20°C responses as our reference response and compare against responses taken at other temperatures. The results of these experiments, shown in Table 6, confirm that RingRAM is robust against even large temperature variations. We expect similar trends to hold for discrete and ASIC implementations.

10 IMPROVING SYSTEM SECURITY

In this section, we explore how system designers use RingRAM to improve overall system security. Specifically, we implement a RISC-V-based Linux system that leverages RingRAM to fix security weaknesses brought on by slow True Random Number Generators (TRNGs) and persistent insecure coding practices: (1) an Internet-of-Things (IoT) system designer adds RingRAM to leverage its high rate of entropy to quickly seed Linux's pseudorandom number generator after power-on and (2) a system designer replaces Linux's (pseudo)random device completely with RingRAM and even poorly-coded software becomes more secure. From a high level, these tests show how valuable RingRAM is to overall system security and how it fits with IoT-class systems.

The full-system prototype is a System-on-Chip (SoC) centered on the 64-bit Rocket RISC-V implementation [2]. The Rocket core connects to peripherals through a 64-bit AXI bus [2]. We create a 64-cell RingRAM module that connects to the processor bus using a AXI-Lite interface [44] that exposes two 32-bit Physical Unclonable Function (PUF) and two 32-bit TRNG registers. The SoC is implemented on the Artix-7 FPGA used in §8, but with a microSD expansion adaptor [14] that we need to store the boot image and other software. Software executes from 256MB DDR3. The resulting SoC consumes 84.6% of the FPGA's LUTs, with RingRAM incurring an additional 0.8% overhead; RingRAM has no effect on power or maximum frequency.

On this hardware platform, we run the busybox [5] user space and Linux 5.5.2. To enable user mode software access to our registers, we create a device driver that uses memory-mapped I/O to access RingRAM's registers and exposes the returned PUF and TRNG responses as a file in the `/proc` file system [24]. For experiments involving RingRAM, we replace Linux's default `/dev/random` and `/dev/urandom` devices with our own device driver. In doing this, **RingRAM services all software requests for (pseudo)random values—increasing security without software modification**.

10.1 Filling the Boot-time Entropy Hole

The IoT era brings with it a new set of security concerns. While many of these concerns are addressable using traditional cryptographic primitives, such primitives require truly random numbers that are infeasible for an attacker to guess. As explained in §2, operational chaos is the only viable source for such numbers. The focused and embedded nature of many IoT devices means that there are few interfaces to operational chaos, dramatically limiting the rate of entropy. Previous work shows the consequence of this slow accumulation of entropy is the use of duplicate and weak keys (e.g., .75% of TLS certs. and 1% of SSH DSA, respectively), because software requires randomness before it is available [19, 20, 25]; this is referred to as the boot-time entropy hole [20].

Test	RingRAM			Linux Default		
	overhead	σ/μ	p-value	overhead	σ/μ	p-value
sha-256	0.13%	0.37%	0.12	0.01%	0.26%	0.39
sha-512	0.00%	0.45%	0.33	0.13%	0.37%	0.15
aes-128	0.52%	1.17%	0.17	-0.13%	1.25%	0.28
aes-192	-0.05%	0.15%	0.33	0.03%	0.20%	0.24
aes-256	0.12%	0.18%	0.06	0.08%	0.23%	0.23
rsa-1024	0.00%	0.00%	-	0.00%	0.00%	-
rsa-2048	0.00%	0.00%	-	0.00%	0.00%	-
average	0.19%	0.34%	0.15	0.00%	0.41%	0.48

Table 7: RISC-V Openssl speed test results

To show that RingRAM effectively fills the boot-time entropy hole, we apply our RingRAM-enhanced system to the findings of the "Mining Ps and Qs" paper. There, the authors show that it takes roughly 66 seconds for 192-bits of true randomness to accumulate so that Linux's random device (`/dev/random`) can seed the pseudorandom device (`/dev/urandom`). Insecurity arises when a security-critical program (e.g., `sshd`) pulls from the pseudorandom device for secret generation before it has been influenced by the random device (e.g., within the first 5 seconds post-boot-up in the case of `sshd`). Without RingRAM, the limited sources of entropy (e.g., clock skew) result in a deterministic result from the pseudorandom device—eliminating security guarantees. Measurements from our SoC show that, with RingRAM, 192-bits of true randomness is available in 77.4μs. Thus, **by the time `sshd` requests data from the pseudorandom device, it could be influenced by the random device >60,000 times**, making the returned results—and resulting key—unique and non-deterministic.

10.2 Taking the pseudo out of random

While */dev/random* provides a source of true randomness, it traditionally comes at the cost of halting the execution of programs that access it while it accumulates sufficient randomness to service the request. This is why operating systems present an alternative interface that provides a best-effort source of apparent randomness; in Linux, programs access this interface through */dev/urandom*. To eliminate blocking, */dev/urandom* turns a small amount of true randomness from */dev/random* into an unbounded amount of apparent randomness. Because the outputs from */dev/urandom* are based—deterministically—on a small amount of true randomness, any security based on them is limited to the length and secrecy of the seed value from */dev/random*. While Linux documentation states that */dev/urandom* must not be used for security-critical applications, many (including `openssl`) use it because of its more convenient interface and due to developer ignorance or distrust of

black-box TRNGs. Work over the years shows that the pseudorandom generators that underpin */dev/urandom* have flaws that limit their security beyond what is expected [10]. Instead of trying to chase the perfect pseudorandom number generator implementation, we create a blocking-free TRNG using RingRAM. This eliminates the need for */dev/urandom*,[15] reducing the trusted computing base and automatically eliminating programmer errors.

To show that our */dev/urandom*-free prototype has no ill-effects on software, we boot Linux and run `openssl`'s built-in benchmark suite `speedtest` on the most popular cryptographic algorithms `SHA-2`, `AES`, and `RSA`, using the most common key sizes. We performed 20 trials of each configuration to account for noise from the operating system and timing measurement. Note that by replacing Linux's (pseudo)random devices, all software that requires randomness uses RingRAM, not just `openssl`. The system and `openssl` perform—problem free—for the entire evaluation. Table 7 provides both the running time of RingRAM-based and the default sources of randomness relative to `openssl`'s CRNG pseudorandom generator. These results show that RingRAM provides a TRNG that is as fast as pseudorandom generators, opening the door to TRNG-only systems.

11 RELATED WORK

While Ring Oscillators (ROs) and Static Random Access Memory (SRAM) are the most common and well-studied foundations for a unified hardware security primitive, researchers have explored other circuits to capture manufacturing and operational chaos. In general, there are two broad classes of approach: digital and analog. ROs and SRAM are digital approaches as they are comprised of digital gates whose output value encodes chaos. Alternatively, a Phase Locked Loop is an analog system that directly measures chaos in the analog domain. For Internet-of-Things deployments that are often highly-constrained and low-cost, digital systems are preferred since they do not require special purpose blocks while building a chip. Here we cover some of the more popular RO and SRAM alternatives.

Self Timed Ring (STR)-based TRNGs [9] and PUFs [31] are an extension to the base RO design, aimed at increasing entropy rate. They replace an inverter with a Muller-C+inverter combination. Muller-C gates are unique in that both inputs must be equal to set or reset its outputs, maintaining its state otherwise. By using dual inputs, multiple propagation loops are created that run simultaneously, allowing them designs to capture operational chaos more efficiently, but with a significantly higher overhead cost compared to ROs—making one of its weak points worse; when area overhead is accounted for, the benefit STRs over ROs disappears.

Phase Locked Loops (PLL) are built-in analog components that provide on-chip clock generation. To guarantee the reliability of the output frequency, PLLs have built-in control circuity that dynamically adjusts a Voltage Controlled Oscillator (VCO) [18]. This mitigates all manufacturing and systematic operational variation, but operational chaos remains as jitter in the output frequency. PLL based TRNGs capture jitter using a series of cascading Flip-Flops running off the PLL's input source, effectively creating a metastable state between the input and output frequencies [16, 38, 48]. Compared to RO's, PLL based TRNGs minimize area overhead by re-utilizing included PLLs. Unfortunately, we are unaware of any PLL-based PUFs and it is not clear how to create one.

Metastability is an unstable equilibrium state in which a devices may persists for an unbound time [8]. Given that operational chaos influences the outcome of a metastable state [42], they are viable foundations for TRNGs. At a high level, both SRAM and RingRAM are metastability based, but their PUF-oriented natures highlights the primary challenge with metastability-based primitives: creating a metastable state. Unlike SRAM and RingRAM which leverage many samples of simple hardware loops, another class of designs are self-tuning in that they dynamically alter circuit parameters to filter systematic variation [26, 29, 39]. While this increases entropy, it adds complexity, and prevents them from serving as a PUF. Another approach is to capture chaos in the time it takes to resolve a metastable state [37], as opposed to the final value. While unique, recent work shows that the metastable settling time is susceptible to systematic operation variation, leaving TRNGs open to attack [7]. RingRAM shows that stable-value focused metastable-base primitives are the right direction, but **choosing between PUFs and TRNGs is a false choice**, a designer can use our enhancements to select the desired balance for their application.

12 CONCLUSION

RingRAM is a hardware security primitive, composed of simple logic gates, that is readily implementable using a range of hardware technologies. Our experiments show that RingRAM provides the necessary foundation for *both* true random number generators and physical unclonable functions. The most important aspect of RingRAM is that it combines the advantages of existing hardware security primitives, but without their drawbacks, making RingRAM more practical, deployable, and secure. We highlight RingRAM's deployability and benefit to system security showing how, when added to a Linux-based System-on-Chip, it increases software security.

The power of RingRAM comes from a focus on chaos, both static and dynamic. A hardware security primitive must harness dynamic chaos to serve as a base for true random number generators. A hardware security primitive must harness static chaos to serve as a base for physical unclonable functions. By understanding available sources of chaos, how they impact hardware, and how existing primitives measure that chaos, it is possible to design a single, ring-based, hardware security primitive to rule them all.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their feedback and suggestions that enhanced the quality of this work. The project depicted is sponsored by the Defense Advanced Research Projects Agency. The content of the information does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred. Approved for public release; distribution is unlimited.

REFERENCES

[1] Ilia A. Bautista Adames, Jayita Das, and Sanjukta Bhanja. 2016. Survey of emerging technology based physical unclonable funtions. In *International Great Lakes Symposium on VLSI (GLSVLSI)*. 317–322. https://doi.org/10.1145/2902961.2903044 ISSN: null.

[2] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee,

[15] We keep it around, because software expects it. It is now just an alias for `/dev/random`

Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator*. Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley.

[3] Pierre Bayon, Lilian Bossuet, Alain Aubert, Viktor Fischer, François Poucheret, Bruno Robisson, and Philippe Maurine. 2012. Contactless Electromagnetic Active Attack on Ring Oscillator Based True Random Number Generator. In *Constructive Side-Channel Analysis and Secure Design*. Vol. 7275. Springer Berlin Heidelberg, Berlin, Heidelberg, 151–166. https://doi.org/10.1007/978-3-642-29912-4_12

[4] Eduardo Boemo and Sergio López-Buedo. 1997. Thermal monitoring on FPGAs using ring-oscillators. In *Field-Programmable Logic and Applications (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 69–78. https://doi.org/10.1007/3-540-63465-7_212

[5] BusyBox. 2019. BusyBox: The Swiss Army Knife of Embedded Linux. https://busybox.net/about.html.

[6] Christian Cachin. 1997. *Entropy measures and unconditional security in cryptography*. Ph.D. Dissertation. ETH Zurich.

[7] Y. Cao, V. Rožić, B. Yang, J. Balasch, and I. Verbauwhede. 2016. Exploring active manipulation attacks on the TERO random number generator. In *International Midwest Symposium on Circuits and Systems (MWSCAS)*. 1–4.

[8] T.J. Chaney and C.E. Molnar. 1973. Anomalous Behavior of Synchronizer and Arbiter Circuits. *IEEE Trans. Comput.* C-22 (Apr 1973), 421–422. https://doi.org/10.1109/T-C.1973.223730

[9] A. Cherkaoui, V. Fischer, A. Aubert, and L. Fesquet. 2013. A Self-Timed Ring Based True Random Number Generator. In *IEEE 19th International Symposium on Asynchronous Circuits and Systems*. 99–106. https://doi.org/10.1109/ASYNC.2013.15

[10] Shaanan Cohney, Andrew Kwong, Shahar Paz, Daniel Genkin, Nadia Heninger, Eyal Ronen, and Yuval Yarom. 2020. Pseudorandom Black Swans: Cache Attacks on CTR_DRBG. In *IEEE Symposium on Security and Privacy (Oakland)*. 750–767.

[11] Ian Cutress and Wendell Wilson. 2020. Testing a Chinese x86 CPU: A Deep Dive into Zen-based Hygon Dhyana Processors. https://www.anandtech.com/show/15493/hygon-dhyana-reviewed-chinese-x86-cpus-amd/3.

[12] Frederik Michel Dekking, Cornelis Kraaikamp, Hendrik Paul Lopuhaä, and Ludolf Erwin Meester. 2005. *A Modern Introduction to Probability and Statistics*. Springer London. https://doi.org/10.1007/1-84628-168-7

[13] Digilent. [n.d.]. Arty A7 Reference Manual. https://reference.digilentinc.com/reference/programmable-logic/arty-a7/reference-manual

[14] Digilent. [n.d.]. Pmod SD Reference Manual. https://reference.digilentinc.com/_media/reference/pmod/pmodsd/pmodsd_rm.pdf

[15] Maurizio Di Paolo Emilio. 2020. EETimes - Maxim Intros MCU with PUF Technology. https://www.eetimes.com/maxim-intros-mcu-with-puf-technology/

[16] Viktor Fischer and Miloš Drutarovský. 2003. True Random Number Generator Embedded in Reconfigurable Hardware. In *Cryptographic Hardware and Embedded Systems - CHES 2002 (Lecture Notes in Computer Science)*, Burton S. Kaliski, cetin K. Koc, and Christof Paar (Eds.). Springer Berlin Heidelberg, 415–430.

[17] Dan Goodin. 2013. We cannot trust Intel and Via's chip-based crypto, FreeBSD developers say. https://arstechnica.com/information-technology/2013/12/we-cannot-trust-intel-and-vias-chip-based-crypto-freebsd-developers-say/.

[18] Guan-Chyun Hsieh and J. C. Hung. 1996. Phase-locked loop techniques. A survey. *IEEE Transactions on Industrial Electronics* 43 (Dec 1996), 609–615. https://doi.org/10.1109/41.544547

[19] Z. Gutterman, B. Pinkas, and T. Reinman. 2006. Analysis of the Linux random number generator. In *IEEE Symposium on Security and Privacy (Oakland)*.

[20] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. 2012. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *USENIX Security Symposium (USENIX Security)*. 205–220.

[21] Daniel E. Holcomb, Wayne P. Burleson, and Kevin Fu. 2009. Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers. *IEEE Trans. Comput.* 58 (Sep 2009), 1198–1210. https://doi.org/10.1109/TC.2008.212

[22] Intrinsic ID. 2017. White Paper - The reliability of SRAM PUF. , 16 pages. https://www.intrinsic-id.com/wp-content/uploads/2017/08/White-Paper-The-reliability-of-SRAM-PUF.pdf

[23] K. Kang, H. Kufluoglu, K. Roy, and M. Ashraful Alam. 2007. Impact of Negative-Bias Temperature Instability in Nanoscale SRAM Array: Modeling and Analysis. *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26, 10 (Oct 2007), 1770–1781.

[24] Michael Kerrisk. [n.d.]. proc(5) - Linux manual page. http://man7.org/linux/man-pages/man5/proc.5.html

[25] JD Kilgallin. 2019. Factoring RSA Keys in the IoT Era. In *IEEE International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications (TPSISA)*.

[26] D.J. Kinniment and E.G. Chester. 2002. Design of an on-chip random number generator using metastability. In *Proceedings of the 28th European Solid-State Circuits Conference*. 595–598.

[27] R. Maes and V. van der Leest. 2014. Countering the effects of silicon aging on SRAM PUFs. In *International Symposium on Hardware-Oriented Security and Trust (HOST)*. 148–153.

[28] Abhranil Maiti, Vikash Gunreddy, and Patrick Schaumont. 2013. A Systematic Method to Evaluate and Compare the Performance of Physical Unclonable Functions. In *Embedded Systems Design with FPGAs*. Springer, New York, NY, 245–267. https://doi.org/10.1007/978-1-4614-1362-2_11

[29] Mehrdad Majzoobi, Farinaz Koushanfar, and Srinivas Devadas. 2011. FPGA-Based True Random Number Generation Using Circuit Metastability with Adaptive Feedback Control. In *Cryptographic Hardware and Embedded Systems – CHES 2011 (Lecture Notes in Computer Science)*. Springer Berlin Heidelberg, 17–32.

[30] Joseph McMahan, Weilong Cui, Liang Xia, Jeff Heckey, Frederic T. Chong, and Timothy Sherwood. 2017. Challenging on-chip SRAM security with boot-state statistics. In *Symposium on Hardware Oriented Security and Trust (HOST)*. 101–105.

[31] Julian Murphy, Maire O'Neill, Frank Burns, Alex Bystrov, Alex Yakovlev, and Basel Halak. 2012. Self-Timed Physically Unclonable Functions. In *5th International Conference on New Technologies, Mobility and Security (NTMS)*. 1–5. https://doi.org/10.1109/NTMS.2012.6208707 ISSN: 2157-4960.

[32] S. P. Park, K. Kang, and K. Roy. 2009. Reliability Implications of Bias-Temperature Instability in Digital ICs. *IEEE Design Test of Computers* 26 (Nov 2009), 8–17. https://doi.org/10.1109/MDT.2009.154

[33] C.S. Petrie and J.A. Connelly. 1996. Modeling and simulation of oscillator-based random number generators. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, Vol. 4. 324–327 vol.4. https://doi.org/10.1109/ISCAS.1996.541967

[34] V. Reddy, A. T. Krishnan, A. Marshall, J. Rodriguez, S. Natarajan, T. Rost, and S. Krishnan. 2002. Impact of negative bias temperature instability on digital circuit reliability. In *International Reliability Physics Symposium (RELPHY)*. 248–254.

[35] A. Roelke and M. R. Stan. 2016. Attacking an SRAM-Based PUF through Wearout. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 206–211.

[36] D. Schellekens, B. Preneel, and I. Verbauwhede. 2006. FPGA Vendor Agnostic True Random Number Generator. In *International Conference on Field Programmable Logic and Applications (FPL)*. 1–6.

[37] Michal Varchola and Milos Drutarovsky. 2010. New High Entropy Element for FPGA Based True Random Number Generators. In *Cryptographic Hardware and Embedded Systems (CHES)*. 351–365.

[38] M. Varchola, M. Drutarovsky, R. Fouquet, and V. Fischer. 2008. Hardware Platform for Testing Performance of TRNGs Embedded in Actel Fusion FPGA. In *18th International Conference Radioelektronika*. 1–4. https://doi.org/10.1109/RADIOELEK.2008.4542712

[39] Ihor Vasyltsov, Eduard Hambardzumyan, Young-Sik Kim, and Bohdan Karpinskyy. 2008. Fast Digital TRNG Based on Metastable Ring Oscillator. In *Cryptographic Hardware and Embedded Systems – CHES 2008 (Lecture Notes in Computer Science)*. Springer Berlin Heidelberg, 164–180.

[40] T.C. Weigandt, Beomsup Kim, and P.R. Gray. 1994. Analysis of timing jitter in CMOS ring oscillators. In *Proceedings of IEEE International Symposium on Circuits and Systems - ISCAS '94*, Vol. 4. 27–30 vol.4. https://doi.org/10.1109/ISCAS.1994.409188

[41] Steve H. Weingart. 2000. Physical Security Devices for Computer Subsystems: A Survey of Attacks and Defenses. In *Cryptographic Hardware and Embedded Systems — CHES 2000 (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 302–317. https://doi.org/10.1007/3-540-44499-8_24

[42] J. Wu and M. O'Neill. 2010. Ultra-lightweight true random number generators. *Electronics Letters* 46 (Jul 2010), 988–990. https://doi.org/10.1049/el.2010.0893

[43] Xilinx. 2016. 7 Series FPGAs Configurable Logic Block User Guide (UG474). , 74 pages.

[44] Xilinx. 2016. AXI GPIO v2.0: LogiCORE IP Product Guide. , 34 pages.

[45] Xilinx. 2018. 7 Series FPGAs Data Sheet: Overview (DS180). , 18 pages.

[46] Xinghai Tang, V. K. De, and J. D. Meindl. 1997. Intrinsic MOSFET parameter fluctuations due to random dopant placement. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 5 (Dec 1997), 369–376. https://doi.org/10.1109/92.645063

[47] S. Zeitouni, Y. Oren, C. Wachsmann, P. Koeberl, and A. Sadeghi. 2016. Remanence Decay Side-Channel: The PUF Case. *IEEE Transactions on Information Forensics and Security* 11, 6 (2016), 1106–1116.

[48] M. Šimka, M. Drutarovský, and V. Fischer. 2011. Testing of PLL-based true random number generator in changing working conditions. *Radioengineering* 20 (2011), 94–101.

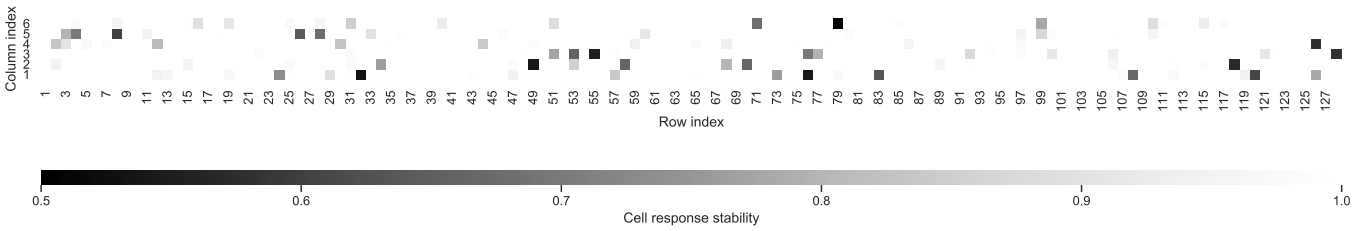

Figure 13: There is no obvious correlation between location and RingRAM cell stability.

A RINGRAM'S SPATIAL LOCALITY

Figure 13 shows the spatial relationship of 768 individual RingRAM cells on a Xilinx Artix-7 FGPA [45] on a Digilent Arty-A7-100T evaluation board [13].

B ARTIFACT APPENDIX

B.1 Abstract

This artifact appendix describes how to use and implement RingRAM to reproduce the results presented in this paper. The artifact contains the source code for RingRAM, an evaluation design, data capture scripts, implementation across three evaluation boards, and is publicly available as a GitHub repository: https://github.com/FoRTE-Research/RingRAM. In addition we demonstrate how to place/rout RingRAM cells across the FPGA as well as porting our evaluation design onto other FPGAs.

B.2 Artifact check-list (meta-information)

- **Program: Vivado, python3**
- **Hardware: Arty-A735, Arty-A7100, Virtex7-VC709**
- **Output: 64-bits depicting the state of 64-RingRAM cells**
- **Experiments: Capture 320,000 samples from 64-RingRAM cells**
- **How much disk space required (approximately): 10MB**
- **How much time is needed to prepare workflow (approximately): 20mins**
- **How much time is needed to complete experiments (approximately): 10mins**
- **Publicly available: Yes**
- **Code licenses (if publicly available): MIT License**

B.3 Description

B.3.1 How to access. This artifact is publicly accessible and cloneable through its GitHub repository: https://github.com/FoRTE-Research/RingRAM. The repository contains the RingRAM primitive, an evaluation design used to test and monitor the state of RingRAM cells, and scripts for creating symmetric and tightly packed cells.

B.3.2 Hardware dependencies. While RingRAM can be ported onto any FPGA system, this artifact was built and evaluated using these three evaluation boards: Arty A7-35, Arty A7-100, and VC709

B.3.3 Software dependencies.

(1) Vivado: All the evaluation boards in this artifact utilize Xilinx FPGAs. Therefore Xilinx's Vivado design suite is required to synthesis, place/route, and generate bitstreams.
(2) python3: As RingRAM layouts are required to be symmetric and tightly packed, this artifact controls placement by modifying the Xilinx Design contraint (.xdc) file. We utilize python to automate this process allowing us to move and port our layout.

B.4 Installation

(1) Clone repository:
  - https://github.com/FoRTE-Research/RingRAM
(2) Creating vivado project using make commands:
  - Arty A7-35: `make RingRAM-A735`
  - Arty A7-100: `make RingRAM-A7100`
  - Virtex 7-VC709: `make RingRAM-VC709`
(3) Creating vivado project manually:
  - Create Project
  - Project Type - RTL Project
  - Add Sources - Include Verilog (.v) files in HDL directory
  - Add Constraints - Include Xilinx Design Constraints (.xdc) files in HDL directory

B.5 Experiment workflow

(1) Generate a bitstream data programming file of the RingRAM evaluation design by running synthesis and implementation on the provided HDL files. This can be done manually or by running make (RingRAM-A735, RingRAM-A7100, RingRAM-VC709)
(2) Capture the serial output using the `captureSerial` script
(3) Download bitstream data programming file into the targeted FPGA device using Vivado's Hardware Manager

B.6 Evaluation

To utilize RingRAM in any design, one need only to instantiate the RingRAM component found in `RRAM.v`. However to evaluate RingRAM we create a state machine (`RRAM_CTRL.v`) that controls the enables of the RingRAM cells and transmits their states through a UART port (`UART_CTRL.v`). Controlling the enables allows us to set and reset their race condition: LOW enables prevents any feedback forcing the outputs to be high, HIGH enables initiates the race condition. To properly evaluate RingRAM cells we must capture and examine the result of multiple race conditions across multiple cells. To achieve this, the state machine continuously toggles the enable and transmits the cell's outputs through the serial. To optimize serial communication we do not encode our data in ASCII, instead the raw

binary values of all the cells are transmitted in bursts. To capture and store these iterations we wrote a script (`captureSerial`) that automatically verifies that when LOW enable all outputs are HIGH and when HIGH enable stores the outputs of the race conditions in a log file.

Command:
`captureSerial [-P] [-F]`
Parameters:

(1) `[-p] [-P] [-PORT]`: Location of the serial port
(2) `[-f] [-F] [-FILE]`: File output path

All results generated in this paper were extracted utilizing this state machine (`RRAM_CTRL.v` on 64 RingRAM cells to capture (`captureSerial`) 320,000 samples. Depending on the desired evaluation there are customization options available: physical placement, inverter chain lengths, and the number of RingRAM cells.

B.7 Experiment customization

There are two sources of variation that a designer must avoid when implementing RingRAM: systematic and structural. We provide two guidelines to avoid these sources of variation: Symmetric and Tightly-packed. The `xdcRingRAMCC` script generates a Xilinx Design Constraints (.xdc) file that adheres to these guidelines:

Command:
`xdcRingRAMCC [-F] [-C] [-I] [-X] [-Y] [-P]`
Parameters:

(1) `[-f][-F][-FPGA]`: Which FPGA design to use
(2) `[-c][-C][-CELLS]`: RingRAM cells to generate
(3) `[-i][-I][-INV]`: Length of the RingRAM inverter chains
(4) `[-x][-X][-POSX]`: Starting horizontal index of the LUT
(5) `[-y][-Y][-POSY]`: Starting vertical index of the LUT
(6) `[-p][-P][-PATH]`: The location and name of the RingRAM primitive

B.7.1 Placement. To customize the physical location of the RingRAM cells modify the [-X] and [-Y] parameters. However, as it is important to keep symmetry, you should examine the LUT placements of the FPGA. Open the synthesis/implementation design in Vivado and examine the layout to determine which cells to use.

B.7.2 Inverter Chain Length. To customize the inverter chain length of the RingRAM cells modify the:

(1) `xdcRingRAMCC`: The `[-I]` parameter
(2) `top_level`: The `g_RRAM_INV` parameter

B.7.3 RingRAM Cells. To customize the number of RingRAM cells modify the:

(1) `xdcRingRAMCC`: The `[-C]` parameter
(2) `top_level`: The `g_RRAM_CELLS` parameter

B.7.4 Porting to another FPGA. The RingRAM primitive `RRAM.v` can be instantiated in any design or on any FPGA. To customize the number of cells or the length of the inverter chains set the `g_RRAM_CELLS` or `g_RRAM_INV` parameters respectively.

However to port the RingRAM layout it is necessary to modify the:

(1) `xdcRingRAMCC`: `[-P]` and `[-F]` parameter
(2) `xdcAddBlocks`: add blocks to contain the equivalent pinouts and pblock locations.