DRNG: DRAM-based Random Number Generation using its Startup Value Behavior

Charles Eckert
ECE Department, Binghamton University
Vestal, NY, USA
(ceckert1)@binghamton.edu

Fatemeh Tehranipoor and John A. Chandy
ECE Department, University of Connecticut
Storrs, CT, USA
(f.tehrani, chandy)@engr.uconn.edu

***Abstract*—True Random Number Generators (TRNG) are used in a variety of applications including cryptographic algorithms, communication systems, simulations, etc. From a security perspective, TRNGs are particularly important because they can produce random output bits that are fully unpredictable and unbiased. Random sources are not often apparent and it is useful to have intrinsic hardware-based random number sources to generate highly random bit sequences. In this paper, we demonstrate the use of DRAM startup values to produce a TRNG. Previous work [1] has shown that DRAM startup values have potential as a physical unclonable function (PUF). However, in our tests using a newer DDR2 DRAM, we demonstrate that the startup values are not suitable for device authentication but they do have use for creating random keys. We have developed an approach to extract random numbers from the DRAM startup values. Finally, we assessed and tested the randomness of selected data by applying the NIST Statistical Test.**

I. Introduction

Electronic hardware systems are a significant part of today's world. These electronic devices are found nearly everywhere and exist with a vast range of designs and purposes. However, the security of these systems has been extremely lacking with little initiative to find and fix weak points before they are discovered. In order to improve the security of these systems, it is useful to have security primitives that can be used to build more complex security algorithms and policies. To the extent possible, we would like these primitives to be provided by the underlying electronics itself since the hardware is less susceptible external tampering and snooping. Over recent years, there has been major effort to use intrinsic circuit variations to develop primitives that produce random behavior - specifically random functions and random number generators. These hardware-based random functions include so-called physically unclonable functions (PUF) [2] [3]. Recent work had demonstrated that certain DRAMs exhibit startup behavior that can be exploited to create a PUF [4], [1]. Specifically, when a DRAM is powered on, the memory cells start up in a seemingly random distributions. These distributions were shown to be suitable for PUF construction. In our work, we attempted to replicate these findings on newer DDR memories, and found that the characteristics expected of a PUF were not evident. While we did find startup behavior in these memories, they were neither random or consistent. In exploring further and with further manipulation, however, we were able to convert these "PUFs" into use as a large source of random bits. The rest of the paper is organized as follows. Section II describes the behavior of DRAM startup values. The background definitions of the random number generations (TRNGs & PRNGs) are illustrated in Section III. Section IV evaluates the DDR DRAM PUF, experimental results in details. We will cover the possible attacks to DRAMs and their application as RNGs in Section V and VI, respectively. Finally, concluding remarks and future works are given in Section VII.

II. DRAM Startup Values

Tehranipoor et al. [1] demonstrated that DRAMs surprisingly have startup values - i.e. non-zero values when the DRAM is powered on. The reason for such behavior is understood by an examination of the typical DRAM cell structure. Figure 1 illustrates the architecture of a typical DRAM memory cell. One side of the capacitor is connected to a voltage of $\frac{Vcc}{2}$, and the other end is initially left floating and then discharged to ground or charged to Vcc when written. The access transistor controls the charging of the capacitor. The difference between the two capacitor plates is read by the bit line to determine the read value. A voltage of Vcc generates the value one and a voltage of 0 gives a zero. At startup, however, before the memory cell is written, the bitline has a voltage of $\frac{Vcc}{2}$, creating unpredictable behavior in the DRAM read values. Thus, the non-zero values at startup. Tehranipoor et al. [1] was able to successfully use this behavior to construct a PUF.

III. Random Number Generation

Random number generators (RNG) are computational or physical devices designed to generate random sequences. They are categorized into pseudorandom number generator (PRNG) and true random number generator (TRNG) groups.

A. Pseudorandom Number Generator

PRNGs do not produce truely random values, instead rely on some form of algorithm to simulate random values. A PRNG relies on using a seed to generate the random value. With access to the seed, the "random" value can be recreated with the same algorithm. Pseudorandom numbers are numbers that appear random, but are obtained in a deterministic, repeatable, and predictable manner.

B. True Random Number Generator

A TRNG, on the other hand, is a device that generates random values based on statistically random events from signal noise to quantum phenomenon. TRNGs often are limited in the number of bits/sec that can be produced. A common strategy involves using the random numbers as seeds in a PRNG to produce more "random" values. A TRNG is a device that outputs a sequence of independent bits. True random numbers are generated in non-deterministic ways. They are not predictable or repeatable. Hardware-based TRNGs utilize physical noise sources given by dedicated hardware such as noisy diodes, thermal noise, radioactive decay, etc. Various types of TRNGs have been proposed using different noise and random sources such as ring oscillator (RO) [5], memory based TRNGs such as flash memory based [6], DRAM remanence based TRNGs [7], etc [8]. TRNGs are crucial for various applications from governmental security, product labeling, gaming, gambling, lotto drawing to cryptography.

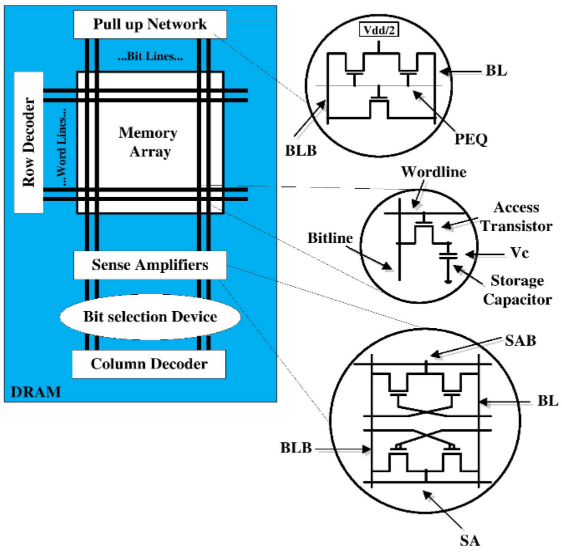
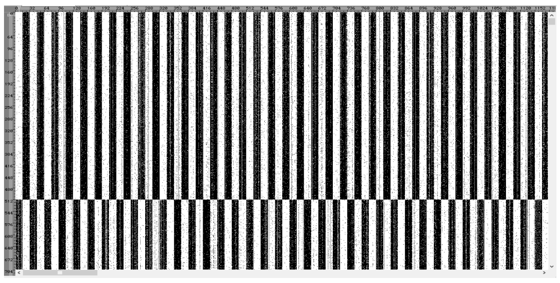
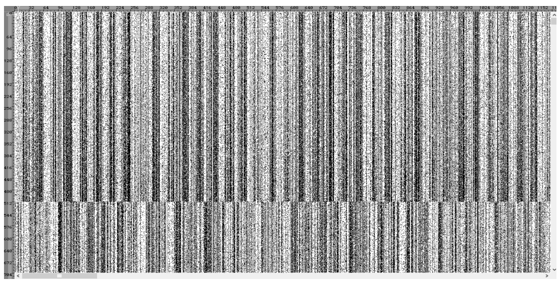


Figure 1: Memory Structure of a One-Transistor DRAM Array



(a)



(b)

Figure 2: Small section of the bitmap from DRAM (a) trial 3, (b) trial 6.

IV. DDR DRAM TRNG Evaluation

A. Experimental Setup

The earlier DRAM PUF work was evaluated using an an older DRAM part, the Hitachi HM51100AL. In our work, we attempted to replicate PUF behavior with newer DDR DRAMs. Our experiment involved using the on-board MIRA P3R1GE3EGF G8E DDR2 on the Diligent Atlys board. (Xilinx Spartan 6 FPGA). The FPGA hardware was configured to transfer the memory data through a serial port at startup. A serial terminal was then used to record the bits and write them to a text file.

B. Data Analysis and Results

The use of newer DRAMs for PUFs has tremendous promise because of the large memory space, but it also contains large potential drawbacks. In our experiments, we found the startup values show a clear bias that reflects the architecture of the DRAM. In addition, not all trials produce adequate results. At times, the startup values are completely non-random yielding no valid data to be used for a PUF. More research is needed to see what is exactly causing the DRAM to start-up to these modes and if it can be avoided. Thus multiple trials will be needed to ensure that the bits are behaving correctly upon startup. We show a graphical representation of a sample bitmap from various trials in Figures 2 (a), and (b). Each row in the Figure represents 8192 bits where white is 0 and black is 1. Across the whole device, clear patterns could be seen when mapping the data to a bit map. The architecture of the DRAM is what is likely strongly influences this style of DRAM. These patterns can be observed across multiple trials. This particular DRAM alternate between ones' and zeros' every 16 bits consistent with the DRAM 16-bit width. Every four megabits, the DRAM has a section of 32 bits before returning to its pattern. Figure 2 (a) shows a small subsection of the bitmap of one of the trials on the DRAM. The pattern described above is heavily noticeable in the figure. However, not every trial gives these results. Some of the trials had far less stable bits and the patterns were less obvious. In Figure 2 (b), you can see the a much larger percentage of bits that don't follow the pattern compared to Figure 2 (a). In Figure 2 (b) despite the higher variance it bits, the original pattern is still easily discernible to the human eye despite the significantly larger number of bits that do not follow the pattern. Of the six trials, Figure 2 (a) appeared to follow the pattern the best while the trial from Figure 2 (b) followed the pattern the least. The other trials were anywhere in-between.

Another interesting result comes when one looks at how the pattern starts. In Figures 2 (a) and (b), one can observe that the pattern starts with 16 bits of zeros first before alternating to 16 bits of ones. However this is not always the case. In other trials, the pattern starts off with 16 bits of ones instead of zeros! So not only is each trial unique and unpredictable with which and how many bits will be affected, a completely inverted pattern can also be achieved. Clearly, the use of DDR DRAM as a PUF is unlikely. With 1 Gbit of memory read, there appears to be no sections on the DRAM that produce both unique and stable bits to be used to generate challenge response pairs. This unpredictability helps us however. Which bits and how many bits that remain stable were extremely unpredictable from trial to trial. Therefore it would be possible to xor multiple trials to introduce the random bits from multiple trials. By xoring multiple trials, we can combine the unpredictability in the bits that do not follow the set patterns. Thus introducing more trials to xor would likely make the results better. It is unlikely that the entire or even a majority of the DRAM can yields random data by xoring multiple trials. However several sections likely will pass randomness tests. The more trials that are included, the more sections will pass. However when xoring 4 different trials together, we get that 1.29% of the blocks pass all the tests. With a 1 Gbit DRAM, this gives us approximately 1.65 MB of random bits that we can use for a secure random code.

1) NIST Statistical Tests Results: We, therefore, use the NIST Statistical Test Suite to assess the randomness in a long sequence of bits [9]. NIST has documented 15 statistical tests, and for each test a p-value should be calculated. p-value $>=$ 0.01 would mean that sequence would be considered to be random with a confidence of 99%. As seen in Table I, xoring the trials gives significantly higher passing rates for some

Table I: DRAM trials and corresponding NIST tests

NIST Test	Trial 1	Trial 2	Trial 3	Trial 4
Frequency	4.99%	7.48%	31.97%	1.13%
BlockFrequency	19.20%	79.84%	95.61%	19.72%
CumulativeSums	4.98%	7.87%	33.29%	1.12%
CumulativeSums	5.28%	8.63%	33.31%	1.23%
Runs	2.35%	0.00%	0.00%	0.09%
LongestRun	16.91%	0.00%	0.00%	5.54%
FFT	0.01%	0.00%	0.00%	0.00%
ApproximateEntropy	0.02%	0.00%	0.00%	0.00%
Serial	27.13%	0.00%	0.00%	0.01%
Serial	95.11%	0.00%	0.00%	5.93%
Pass all tests	0.00%	0.00%	0.00%	0.00%
	Trial 5	Trial 6	Trial 1,3 xor	Trial 1,4,5,6 xor
Frequency	9.20%	0.20%	0.00%	13.12%
BlockFrequency	35.29%	88.92%	0.02%	19.55%
CumulativeSums1	9.42%	0.20%	0.00%	13.19%
CumulativeSums2	9.71%	0.21%	0.00%	13.24%
Runs	0.00%	0.13%	0.00%	18.84%
LongestRun	0.19%	0.18%	0.78%	27.38%
FFT	0.00%	0.41%	1.96%	75.42%
ApproximateEntropy	0.00%	0.00%	0.14%	26.14%
Serial1	0.00%	0.00%	29.95%	64.87%
Serial2	0.02%	0.07%	91.88%	95.41%
Pass all tests	0.00%	0.00%	0.00%	1.29%

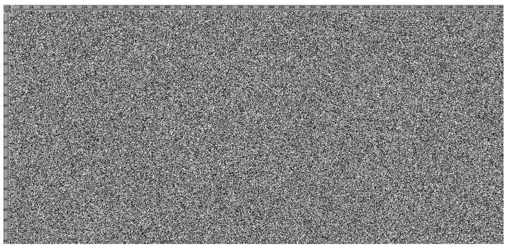


Figure 3: Bitmap from applying Von Numen Corrector on XOR trials 1,2,3,4,5,6

of the later tests that were producing insufficient number of passing blocks. More importantly when you look at the percent of passing all tests, we finally achieve blocks that pass every test.

2) Van Neumann Corrector: Another approach is to apply the Von Neumann Corrector [10] on all the pairs of bits. This works by discarding all "00" and "11" pairs and correcting a "10" to a 1 and a "01" to a 0. Since such a large part of the data has 16 bits of ones or zeros in a row, we can essentially remove such sections and only look at the unstable bits. After applying the corrector, we again used blocks of 65536 bits, and we tested the maximum number of blocks allowed by the new block size. The remaining bits that did not fit into a full 65536-bit block were ignored. Discarding these bits reduced the number of random bits available. Table II shows the result of using the Von Neumann approach. As expected, the amount of data that is available to select a random key is greatly reduced. Of the six trials, the largest key space is 35.1MB or 27.4% of the original DRAM and with the smallest at 9MB or 7.0% of the original DRAM. However, the approach shows some promise with respect to randomness in that it does significantly better than the raw trial results. Unfortunately, it fails the vast majority of the runs tests. Either the bits are changing alternating between runs of ones and zeros too often or not often enough. Since the pattern switches every 16 bits, trials with a large number of unstable bits might alternate too slowly because if it has too many outliers in a 16 bit section. This is supported by looking at the data size of the Von Neumann corrector output. The two smallest datasets (Trials 2 and 3) had the best performance on the limiting test, the runs test.

Table II: DRAM trials with Von Neumann corrector

	Trial 1	Trial 2	Trial 3
Data size	35.1MB	12.8MB	9.0MB
Frequency	97.09%	75.91%	71.97%
BlockFrequency	99.98%	99.39%	98.08%
CumulativeSums	97.68%	78.05%	74.11%
CumulativeSums	97.74%	77.80%	74.05%
Runs	0.11%	1.94%	12.31%
LongestRun	37.06%	60.08%	77.95%
FFT	96.48%	98.16%	98.54%
ApproximateEntropy	19.46%	51.07%	72.22%
Serial	95.91%	97.20%	98.22%
Serial	98.64%	98.78%	98.86%
Pass all tests	0.0350%	0.6248%	5.5974%
	Trial 4	Trial 5	Trial 6
Data size	28.3MB	24.0MB	34.6MB
Frequency	93.46%	89.85%	96.21%
BlockFrequency	100%	99.92%	99.99%
CumulativeSums	94.62%	91.50%	97.05%
CumulativeSums	94.66%	91.48%	96.95%
Runs	0%	0.02%	.02%
LongestRun	24.22%	17.76%	29.13%
FFT	96.40%	96.24%	95.47%
ApproximateEntropy	16.81%	11.14%	11.39%
Serial	27.13%	94.95%	94.90%
Serial	95.11%	98.62%	98.58%
Pass all tests	0.00%	0.0085%	0.00%

This is important to note because the Von Neumann corrector is designed to remove sections of the data that do not have significant number of outliers. Thus, it seems that with less outliers they are more random. Perhaps the more unstable bits there are, the more predictable they become. We come to this conclusion because without any unstable bits, the entire data set would be removed. Thus, the more unstable bits, the larger the trials will be after applying the Von Neumann corrector. Because the smaller trials performed better, its possible that as more unstable bit show up, they are heavily biased in locations leading to a less random result.

A greater number of outliers or unstable bits leading to less random data after the Von Neumann correction is somewhat unexpected. However, this interesting paradox can be solved by combining multiple trials by xoring them. If sections of the unstable bits become predictable, it will be canceled out by the unstable bits from other trials. Looking at results of xoring trials and then performing Von Neumann corrector (Table III), we see a much higher pass rate. The resulting data were all very similar in size with all of them falling in the 25-27% of the original file. All trials had at least 56% of trials pass every test. This is a huge improvement upon earlier results. The best trial for just using the corrector resulted in 5.6% of trails passing and xor'ed trials best resulted in 1.29%.

3) An Overview of Combining Two-methods: Combining these two methods clearly can be used to generate random keys from the unstable bits. Our results can generate at a minimum 15.12 % and a maximum 20.45% of usable random bits. With 1 Gbit of memory to choose from, 15-20% is a sufficiently large enough to generate a secure random key. Also with xoring the trials, we had to xor 4 different trials before getting a small amount of bitstreams to pass. Although xoring more trials produced better results, with this method we saw sufficient results from just xoring two different trials.

As demonstrated earlier, the DRAMs have patterns in there startup behavior. Moreover, these patterns are not consistent, as they differ greatly from trial to trial. It is possible that the DRAM has a certain number of startup states or "modes" that it can enter. Comparing the different trials, two of them had 94.14% of the bits in common. It is entirely possible that the remaining ≈6% is just unstable bits from this particular startup "mode". Supporting the theory on startup modes the

Table III: DRAM trials XOR with Von Neumann corrector

	Trial 1,2 xor	Trial 1,3 xor	Trial 1,2,4,5 xor
File size	34.4MB	33.3MB	33.9MB
Frequency	94.71%	94.73%	95.61%
BlockFrequency	99.43%	99.32%	99.11%
CumulativeSums	95.33%	95.34%	96.08%
CumulativeSums	95.31%	95.21%	96.11%
Runs	80.67%	85.13%	93.80%
LongestRun	94.34%	96.20%	97.70%
FFT	98.53%	98.69%	98.77%
ApproximateEntropy	80.87%	85.70%	89.95%
Serial	98.49%	98.55%	98.70%
Serial	98.94%	99.00%	98.90%
Pass all tests	56.29%	62.61%	74.81%
	Trial 1,3,4,6 xor	Trial 1,4,5,6 xor	Trial 1,2,3,4,5,6 xor
File size	33.1MB	33.0MB	33.6MB
Frequency	93.42%	93.80%	95.02%
BlockFrequency	99.42%	97.35%	98.51%
CumulativeSums	94.25%	94.46%	95.55%
CumulativeSums	94.19%	94.34%	95.41%
Runs	82.90%	95.82%	97.80%
LongestRun	95.40%	98.26%	98.66%
FFT	98.67%	98.90%	98.74%
ApproximateEntropy	85.49%	91.13%	91.92%
Serial	98.64%	98.76%	98.88%
Serial	98.98%	99.06%	98.96%
Pass all tests	59.70%	73.40%	77.92%

16-bit pattern sometimes starts bits set to one and sometimes the bits set to zero. This means a section of mostly stable bits can be all ones one trial and all zeros the next, making it highly likely that some sort of modes exist. More testing is needed to see if reason for the inconsistent numbers and distribution of unstable bits can be determined. The DRAM having a startup mode: would not completely hinder its application. It would reduce the number of challenges, but an attacker would have no way to know which mode it is using and would be unable to recreate the seed state. If the mode can be determined the DRAM would be less secure for generating random numbers. This would allow an attacker with physical access to replicate the seed state. Furthermore all of this behavior is untested and unconfirmed in other DRAMs of the same type. More testing is needed to determine if the startup values are based on modes and if other DRAM have modes and if so do the modes behave the same as the DRAM we tested. Despite these challenges, the DRAM might still have application in cyber security.

V. Possible Attacks

Tests will need to be preformed on more devices to see if they a perform similarly. If all DDR DRAM behave in this way, the only way for an attacker to reproduce a key would be with physical access to the device that was used to generate the key. This ability to reproduce is also limited with the varying amount of amounts of unstable bits in each trial. Further if we choose to xor the trials and apply the Von Neumann corrector, it would become even harder to for an adversary to reproduce the results without knowing the details of each trial that was xor'ed. It is possible for an attacker with knowledge of the memory architecture to probe a DRAM and learn the factors that cause the varying amount of unstable bits. However, this would have little benefit to the attacker, as there is no standard for error in a key. An attacker correctly predicting 90% of the bits won't help as one wrong bit is all that is needed to deny the attacker access.

VI. Application as RNG

The DRAM we tested generated 3350 random bitstreams with a length of 65536 bits. While our results yield a large number of usable random bits, DRAM memory access is relatively slow and we used multiple trials to generate the random bitstreams. This means that our model does not produce random bits at a high rate. The random bits can be used as seeds in an PRNG to produce more random keys. Since the DRAM randomness comes from physical intrinsic introduced by the manufacturing process, it might be possible for an attacker to reproduce sections of the random key if they have access to the device. This makes our model useful for one time uses. The random bitstreams can be extracted and afterwards, the DRAM can be destroyed or rendered in an unusable state to prevent an attacker from attempting to reproduce the bitstreams. Since DRAM is present in almost all modern computer systems and inexpensive, most people will have access to a device that contains DRAM. The availability of DRAM combined with the large number of bits available makes DRAM an excellent candidate for a practical TRNG. However since DRAM is used as main memory and random values are generated on startup, the computer system needs to be shutdown before the DRAM can be used as a TRNG. Once information is loaded into the DRAM, the DRAM will need to be powered down to allow the capacitors to discharge. Since main memory is needed for any computer system to function, normal operation of the computer must be suspended on startup to prevent data being loaded into main memory. Our model relies on transferring the random bits from the DRAM to a separate non volatile memory so that normal operation of system can resume and to prevent losing the random bits in the event of a power-failure.

VII. Conclusion and Future Work

While older DRAMs may show potential as a PUF, our work with modern DDR DRAMs show that they not satisfy criteria to serve as a PUF. Specifically, these startup value patterns were neither random or reliable. However, we have shown that DDR DRAMs can still be used to generate random numbers. Using a variety of correction mechanisms, we are able to improve the randomness of the numbers such that they pass the NIST tests. Further work will investigate the suitability of the approach on a variety of DRAM parts.

References

[1] F. Tehranipoor *et al.*, "DRAM-based intrinsic physically unclonable functions for system-level security and authentication," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2016.

[2] B. Gassend *et al.*, "Silicon physical random functions," in *Proceedings of the 9th ACM conference on Computer and communications security*. ACM, 2002, pp. 148–160.

[3] F. Tehranipoor *et al.*, "Dram pufs reliability analysis due to device accelerated aging," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2017.

[4] ——, "DRAM based intrinsic physical unclonable functions for system level security," in *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*. ACM, 2015, pp. 15–20.

[5] S. N. Dhanuskodi *et al.*, "A chaotic ring oscillator based random number generator," in *Hardware-Oriented Security and Trust (HOST), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 160–165.

[6] Y. Wang *et al.*, "Flash memory for ubiquitous hardware security functions: True random number generation and device fingerprints," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 33–47.

[7] F. Tehranipoor *et al.*, "Robust hardware true random number generators using dram remanence effects," in *Hardware Oriented Security and Trust (HOST), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 79–84.

[8] ——, "A study of power supply variation as a source of random noise," in *2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID)*, Jan 2017, pp. 155–160.

[9] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," DTIC Document, Tech. Rep., 2001.

[10] S.-H. Kwok *et al.*, "A comparison of post-processing techniques for biased random number generators," in *IFIP International Workshop on Information Security Theory and Practices*. Springer, 2011, pp. 175–190.