



In-situ Extraction of Randomness from Computer Architecture through Hardware Performance Counters

*Manaar Alam, Astikey Singh, Sarani Bhattacharya, Kuheli Pratihar,
and Debdeep Mukhopadhyay*

Secure Embedded Architecture Lab(SEAL)
Indian Institute of Technology, Kharagpur



CARDIS 2019
18th Smart Card Research and Advanced
Application Conference

Introduction

- TRNGs are essential building blocks of modern embedded security systems
- Enables various cryptographic algorithms, protocols and secured implementations
- True randomness cannot be obtained via computational methods
- TRNGs derive its randomness from physical parameters
- Security relies on the unpredictability and uniformity of the random numbers

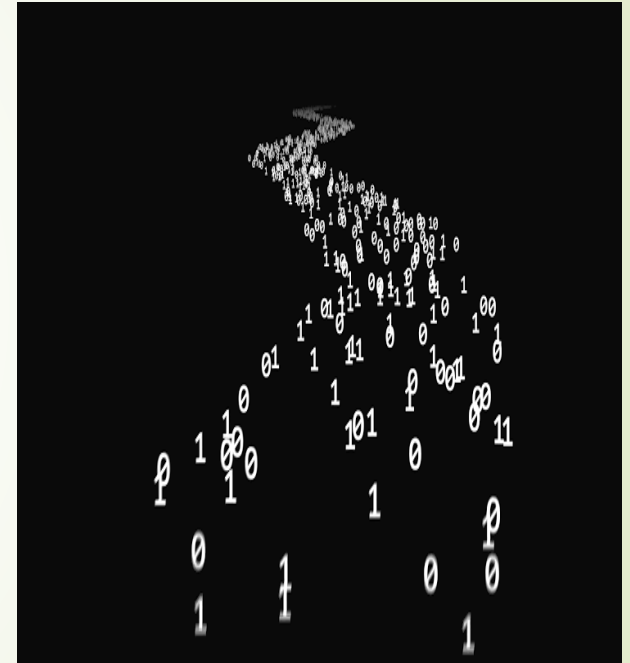
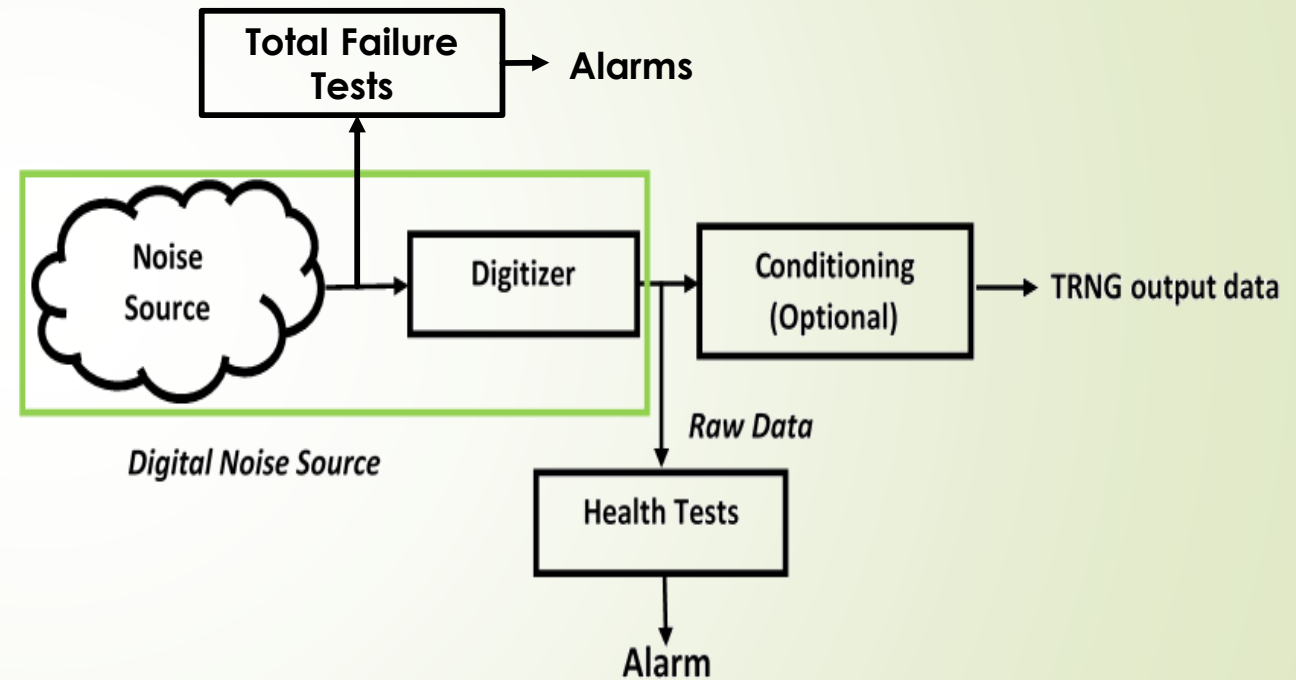


Image Source: Google Image

Generic Architecture of a TRNG

- **Entropy/ Noise Source:** The only component with non-deterministic behaviour
- **Digitization Module:** Converts analog signals into a digital form
- **Post-processing:** Improves the statistical and security characteristics of the raw random numbers
- **Online tests:** Detects failure in generating raw random numbers
- **Total Failure Tests:** Implemented for the fast detection of the total breakdown of the entropy source



Traditional TRNG Designs

Thermal noise, also known as Johnson–Nyquist noise [68], is intrinsic electronic noise which occurs regardless of any applied voltage.

[1] A low power, low voltage Truly Random Number Generator (TRNG) for EPC Gen2 RFID tag was proposed and realized in in **SMIC 0.18 μm standard CMOS process**

[4] Presents the design of a mixed-signal **RNG IC** suitable for integration with hardware cryptographic systems

Metastability is the most commonly used entropy source for both FPGA and ASIC TRNGs. TRNGs of this type rely on the circuit symmetry to achieve unbiased outputs.

[3] Utilizes the write collisions in Block Memory (BRAM)s of TRNGs as entropy sources. *Due to the lack of the low-level understanding of BRAM, as it is a company secret, it is almost impossible to characterize the randomness-generating process and to evaluate its security.*

[5] The last passage time of ring oscillators is utilized as the entropy source. Fabricated in **0.13- μm CMOS technology**

Timing Jitter is defined as the deviation from a periodic signal, such as a reference clock signal

[2] Exploits the jitter of events propagating in a self-timed ring (STR) to generate random bit sequences at a very high bit rate. *Implemented using Altera and Xilinx FPGA*

[6] TRNG based on high-precision edge sampling. *Implemented using Xilinx Spartan 6 and Intel Cyclone V FPGAs*

CMOS Designs:

- Not preferred for high speed applications
- Not easily portable to FPGA families

OUR WORK

- ❖ **Source of Randomness:** Underlying Hardware Architectural Events
- ❖ No external hardware i.e. **SoC design**

TRNGs implemented using external hardware are susceptible to physical attacks

[7] Presents a contactless and local active attack on ring oscillators (ROs) based TRNGs using electromagnetic fields. It is possible to lock them on the injected signal and thus to control the monobit bias of the TRNG output even when low power electromagnetic fields are exploited

[8] A frequency injection attack which is able to destroy the source of entropy in ring-oscillator-based true random number generators (TRNGs).

On- the fly testing of TRNGs

A design methodology for embedded tests of entropy sources.

[9] The proposed solution uses canary numbers which are an extra output of the entropy source of lower quality. This enables an early-warning attack detection before the output of the generator is compromised.

[10] Design of on-the-fly tests based on the attack effects. Uses an empirical design methodology consisting of two phases: collecting the data under attack and finding a useful statistical feature.

- ❖ It would be desirable to develop TRNG sources which are available to a program without resorting to an external component
- ❖ In-situ TRNG design would also make physical attacks more challenging

NIST SP 800-22 STATISTICAL TEST SUITE

Test Type	Defect Detected
Frequency(Monobit)	Too many zeros or ones
Frequency within a block	Too many zeros or ones in specific block sizes
Runs	Too many (or too few) runs of zeros or ones
Longest run of ones in a block	Too many long runs of ones in specific block sizes
Binary Matrix Rank	Linear dependence among fixed length substrings of original
Spectral	Periodic features in the bitstream
Non-overlapping template matchings	Too many occurrences of non-periodic templates
Overlapping template matchings	Too many or too few occurrences of runs of ones
Maurer's Universal Statistical	Too easy to compress bitstream without loss of information
Linear Complexity	Sequence not complex enough to be considered random
Serial	Non-uniform distribution of specific length words
Approximate Entropy	Non-uniform distribution of specific length words
Cumulative Sums	Random walk excursions away from zero too large
Random excursions test	Too many visits of a random walk to a certain state
Random excursions variant	Too many total visits (across many random walks) to a certain state

AIS 20/31 TESTS

Procedure A		test0 is executed once on a 65536*48 bit sequence followed by 257 repetitions of test0 through test5 on successive 20000 bit sequences
test0	disjointedness test	65536 48-bit strings are collected, sorted. No two adjacent values should be equal
test1	monobit test	The number of ones must be between 9654 and 10346
test2	poker test	Distribution of 4 bit tuples checked for 15 degrees of freedom
test3	runs test	Runs of 1, 2, 3, 4, 5, and 6 ones and zeros are checked for expected occurrences
test4	longest run test	No single run can be larger than 34
test5	auto-correlation test	The overlap of the bit stream in the latter half of the sequence is compared to the sequence with the largest overlap in the first half of the sequence.
Procedure B		Distribution tests are conducted for widths of 1, 2, 4, 8 bits on successive samples followed by a single repetition of test 8 on a 256000 + 2560 bit sequence. Total sample size is depends on sample content.
test6	uniform distribution test	Test6a is a monobit test to ensure the number of ones is between 25% and 75% of total. Test6b is a special case of test 7 with a width of 2.
test7	homogeneity test	Collect 10,000 occurrences of runs less than the given width and check for the expected transition probabilities. Test7a corresponds to a width of 3, Test7b corresponds to a width of 4.
test8	entropy estimation (Coron's test)	Accumulate the nearest predecessor distance between byte values in a 256000 + 2560 bit sequence and calculate the empirical entropy.

Contributions

- ▶ TRNG derived from computer architecture, which thrives on the randomness observed through the Hardware Performance Counters. HPC event counters provide a cumulative count to the architectural events and thus proposed to be a high source of entropy.
- ▶ It was also observed that the randomness was highest in the Least Significant bits (LSBs) for the observed values from these counters.
- ▶ These event counter statistics over the monitored application along with the background noise can only be observed at periodic intervals. In order to increase the throughput of the overall random number generation, we pair the proposed TRNG with a secured hash implementation using the Keccak algorithm.

Hardware Performance counters

- Set of special purpose registers, present in most of microprocessor's Performance Monitoring Unit(PMU)
- Store hardware and software events related to the execution of a program, such as cache misses, retired instructions, retired branch instructions, etc.
- Type and number of hardware interrupts vary across different Instruction Set Architectures(ISA)
- Various open-source tools can measure this HPC values: perf tools, PAPI, Oprofile, Valgrind and many more

COMMAND LINE LINUX TOOLS

- *perf* : accesses and reads the HPC registers through the perf event system call for Linux versions above 2.6.31.

Syntax:

```
perf stat -e <event name> -I <interval duration>  
<executable name>
```

- *mpstat*: a utility that collects and displays information about CPU utilization and performance statistics.
/proc/interrupts records the number of interrupts per IRQ on the x86 architecture.
- *taskset*: sets or retrieves the CPU affinity of a running process given its PID (Process ID)

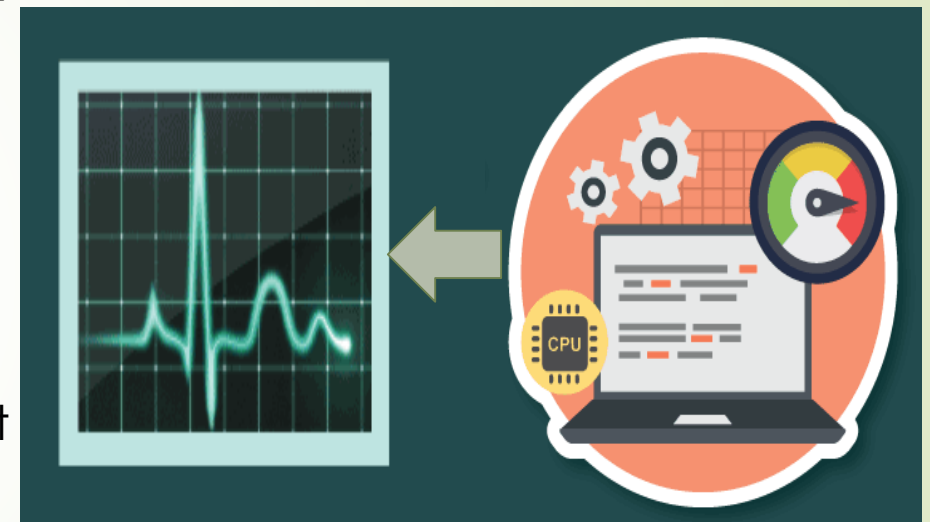
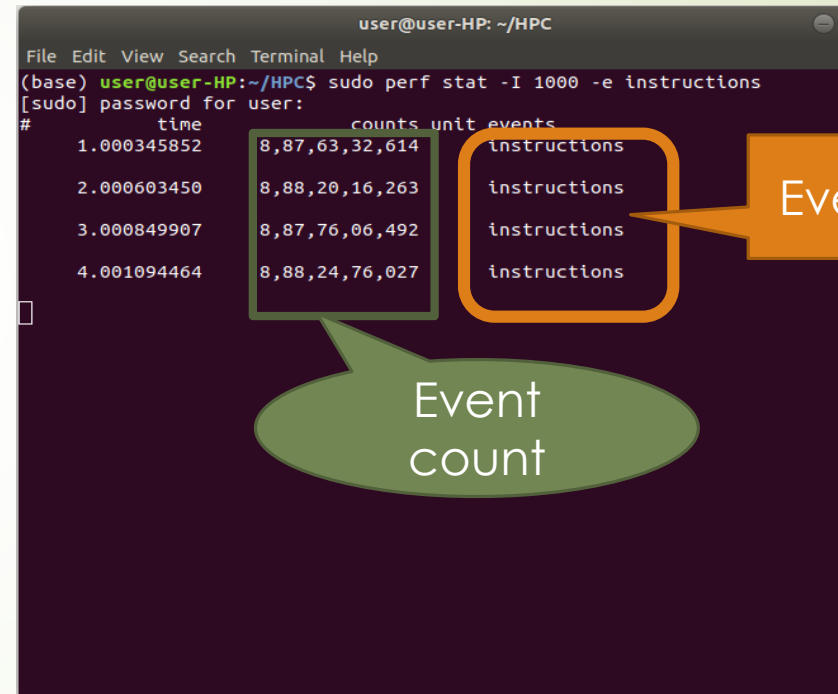


Image Source: Google Images

Monitoring the HPCs

- An infinite loop C code snippet was taken and allowed to run indefinitely
- Various event counts such as instructions, bus-cycles etc. were observed
- Measured the total number of interrupts received per second
(Note: The experiment was performed on a per-core approach)



```
user@user-HP: ~/HPC
File Edit View Search Terminal Help
(base) user@user-HP:~/HPC$ sudo perf stat -I 1000 -e instructions
[sudo] password for user:
#      time          counts unit events
1.000345852 8,87,63,32,614 instructions
2.000603450 8,88,20,16,263 instructions
3.000849907 8,87,76,06,492 instructions
4.001094464 8,88,24,76,027 instructions
```

#	time	counts	unit	events
1.000345852		8,87,63,32,614	instructions	
2.000603450		8,88,20,16,263	instructions	
3.000849907		8,87,76,06,492	instructions	
4.001094464		8,88,24,76,027	instructions	

NON-DETERMINISM IN HPCs

Source of Non-Determinism:

Hardware Interrupts

Ideal case: The HPC events instructions and cpu-cycles should report constant values over the duration of time

Observation: The number of instructions and the number of CPU cycles is not constant over time

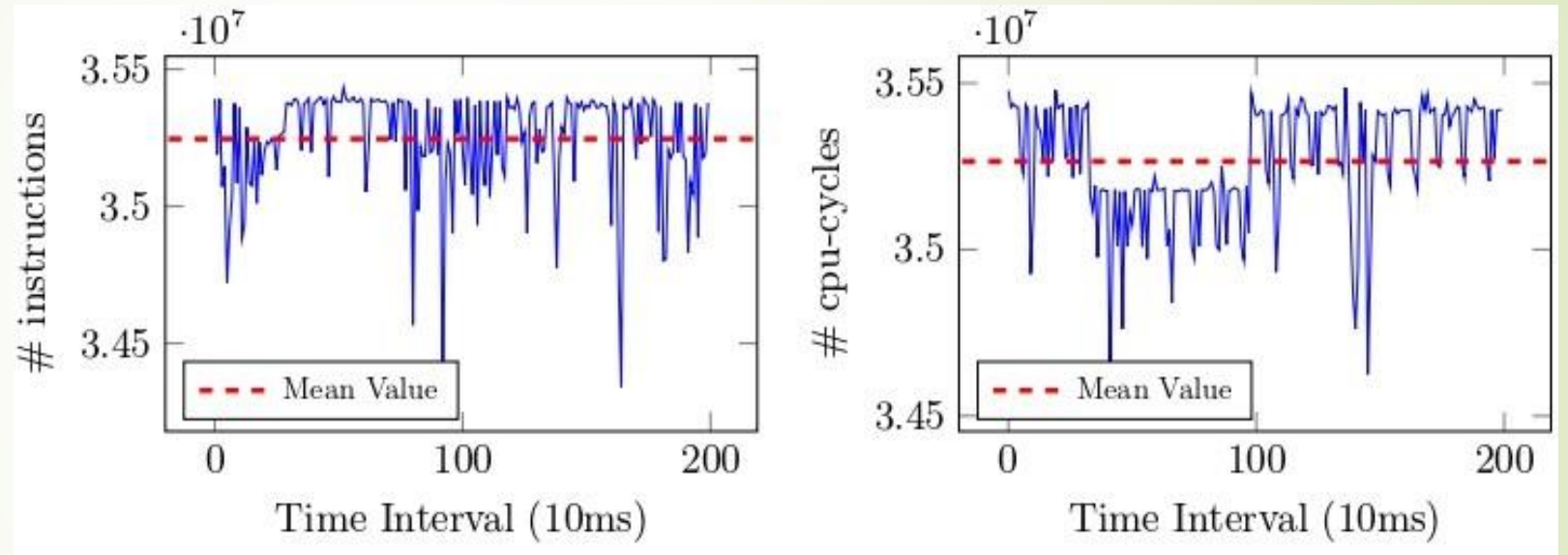


Fig: Performance counter events (i) instruction and (ii) cpu-cycles over the executable of infinite loop with 10ms* interval of time

Significant amount of non-determinism is exhibited by these performance counters

EFFECT OF HARDWARE INTERRUPTS ON HPC EVENTS

Observation:

Whenever there is a surge in the number of interrupts, the count of the events also increase



Validation:

There exists an association between hardware interrupts and HPC events.

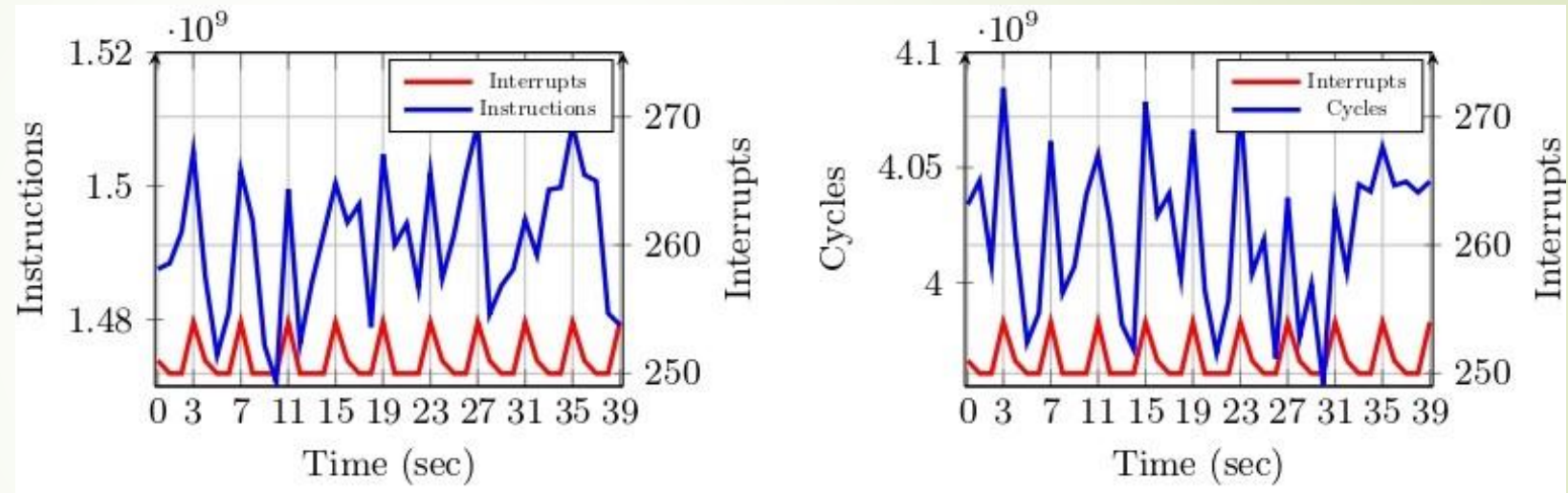
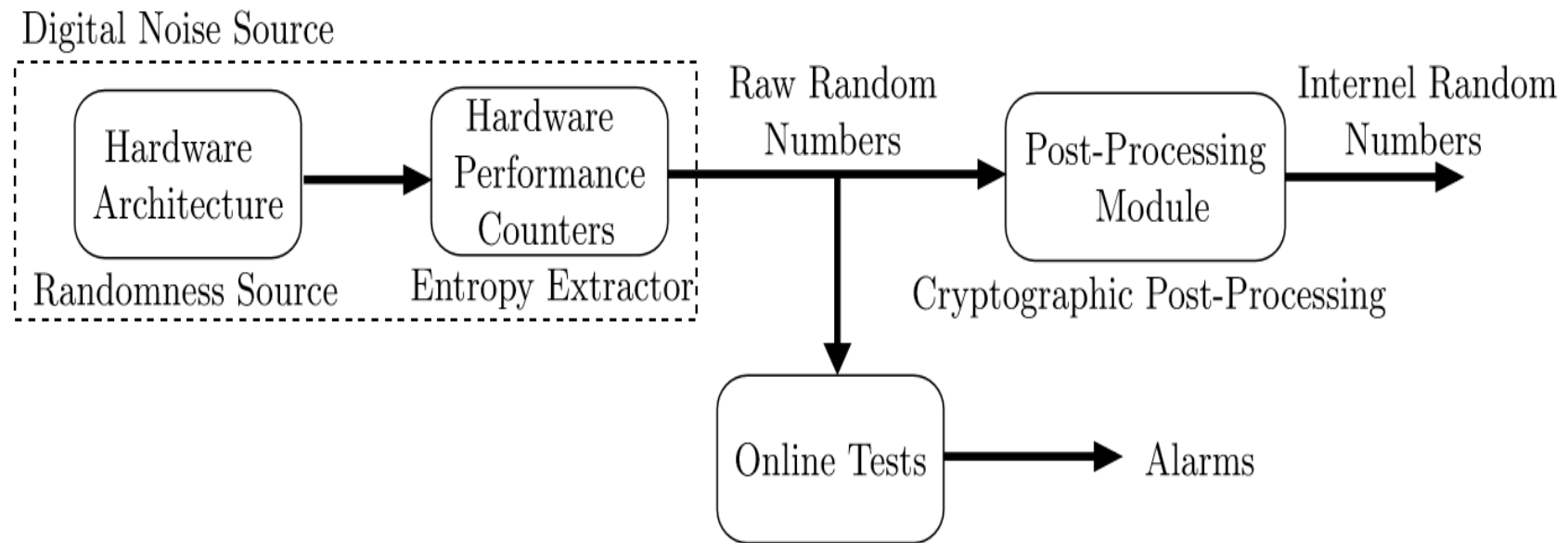


Fig: Effect of hardware interrupts on the HPC events (i) instructions and (ii) cpu-cycles monitored over an infinite loop on different time instances

There are several types of interrupts affecting these HPC events such as Local Timer Interrupts (LOC), IRQ Work Interrupts (IWI), Rescheduling Interrupts (RES), Function Call Interrupts (CAL), and TLB Shootdowns (TLB). The effect of these interrupts can be monitored efficiently using `/proc/interrupts`

PROPOSED TRNG DESIGN



RANDOMNESS EXTRACTION USING HPCS

❖ Selection of Least Significant Bits

- Observed 500,000 instances of the performance counter events *instructions* and *cpu-cycles*, and calculated the entropy for each bit position.
- **Entropy** of **each bit position is not same** for the binary sequences of the monitored values
- Entropy is **highest with LSB** while MSB is highly predictable
- Transformation of the data to binary sequences and considered the last 9[†] bits for further analysis

†: We empirically selected last 9 least significant bits for our experimental setup as for most of the events the last 9 bits provide highest entropy values

Choosing bits from the LSB

❖ Selection of Least Significant Bits(Contd.)

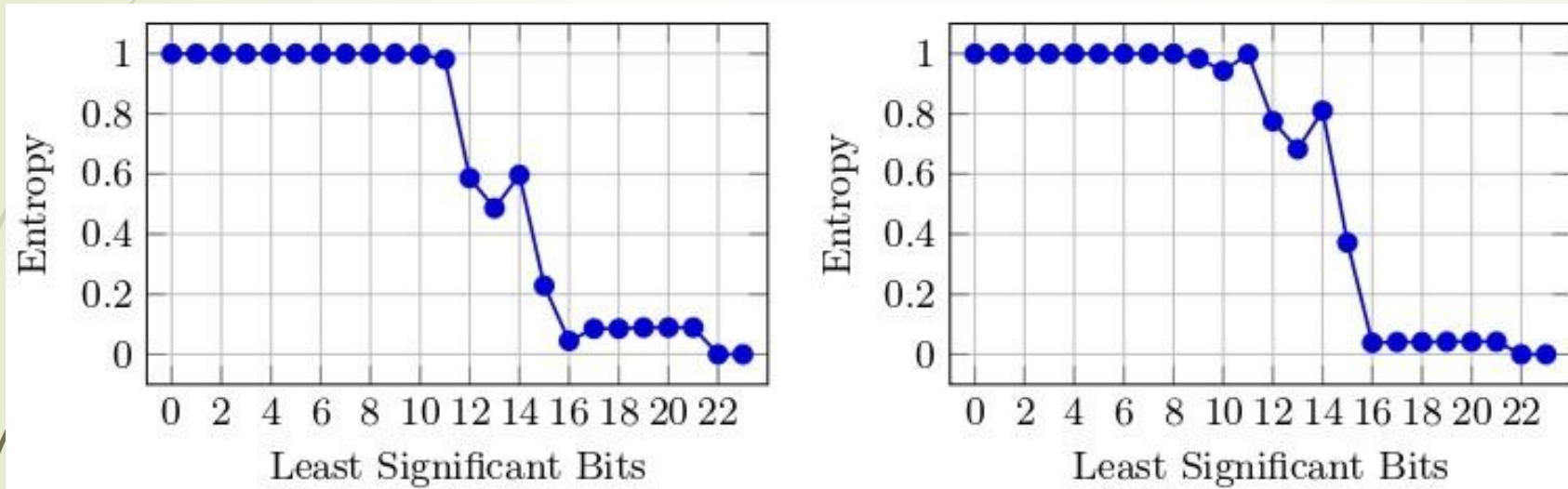


Fig: Entropy of each LSBs for HPC event (i) instructions, and (ii) cpu-cycles

Observation:
LSBs have the highest entropy, and as we move towards the MSBs, the entropy gets reduced

Next-bit Test for HPC events

❖ Selection of HPC events using Yao's Next-Bit Test

- In order to estimate the probability, N sequences of an HPC event at N successive intervals of time were considered
- Given first m -bits of the n possible bits for any sequence $S(n, t)$, i.e., the sequence $S(m, t)$ is already given (where $m < n$)
- According to Yao's Next Bit test,

The sequence $S(n, t)$ has no bias if probability of the $(m + 1)^{th}$ bit being zero is $0.5 \pm \delta$ (i.e., $[Pr_m^t = 0] = 0.5 \pm \delta$), given the knowledge of $S(m, t)$, when δ is negligible (with respect to the security parameter).

- No. of possibilities for $S(m, t) = 2^m$

❖ Selection of HPC events using Yao's Next-Bit Test (Contd.)

- Consider the case $m=4$, i.e. first 4 bits of the binary sequence is known
- Observed $N = 500,000$ values for the events and estimated the probability

2⁴ possible combinations

- If the first 4 bits are 0000, then the estimated probability that the next bit will be 0 is 0.499362 with a bias of 0.000638

Known Bits	Estimated Value of $\hat{\Pr}[b_4^t = 0]$				Value of δ			
	Hardware Performance Counter Events				Hardware Performance Counter Events			
	instructions	cpu-cycles	cache-misses	branches	instructions	cpu-cycles	cache-misses	branches
0000	0.499362	0.499119	0.483038	0.511926	0.000638	0.000881	0.016962	0.011926
0001	0.500616	0.498508	0.510286	0.5	0.000616	0.001492	0.010286	0
0010	0.50388	0.499933	0.61523	0.473591	0.00388	0.000067	0.11523	0.026409
0011	0.503006	0.501612	0.538575	0.472271	0.003006	0.001612	0.038575	0.027729
0100	0.497589	0.500212	0.465892	0.494755	0.002411	0.000212	0.034108	0.005245
0101	0.501385	0.503288	0.499264	0.489194	0.001385	0.003288	0.000736	0.010806
0110	0.497944	0.499307	0.49388	0.480069	0.002056	0.000693	0.00612	0.019931
0111	0.497515	0.498644	0.545499	0.529411	0.002485	0.001356	0.045499	0.029411
1000	0.501878	0.497065	0.532874	0.480286	0.001878	0.002935	0.032874	0.019714
1001	0.509205	0.500564	0.325212	0.473333	0.009205	0.000564	0.174788	0.026667
1010	0.503668	0.498804	0.588985	0.507633	0.003668	0.001196	0.088985	0.007633
1011	0.500938	0.500415	0.345577	0.476785	0.000938	0.000415	0.154423	0.023215
1100	0.49932	0.504391	0.681509	0.483871	0.00068	0.004391	0.181509	0.016129
1101	0.499705	0.499179	0.578446	0.470919	0.000295	0.000821	0.078446	0.029081
1110	0.502052	0.501125	0.357142	0.477891	0.002052	0.001125	0.142858	0.022109
1111	0.500587	0.497146	0.437479	0.481415	0.000587	0.002854	0.062521	0.018585
Average δ					0.002236	0.001493	0.073995	0.018411

Events **instructions** and **cpu-cycles** can act as better candidate for source of randomness

Table: Next-bit test for different HPC events for $m = 4$

Experimental Validation

❖ Results on TRNG output obtained from HPC Events

- Experiments were conducted on two different processors
- Access to HPC events is available to users with administrative privilege
- Primitive events such as **instructions, cpu-cycles, bus-cycles, cache-misses, branches** etc. were considered.

Processor	Linux Version
AMD A10-8700P Radeon R6	Ubuntu with Kernel 4.13.0-36
Intel Core i7-7567U	Ubuntu with Kernel 4.15.0-33

Table: Experimental Setup for Validation of the proposed claim

EXPERIMENTAL VALIDATION(Contd.)

NIST Test	Intel			AMD		
	instructions	cpu-cycles	cache-misses	instructions	cpu-cycles	cache-misses
<i>Frequency</i>	PASS	PASS	FAIL	PASS	PASS	FAIL
<i>BlockFrequency</i>	PASS	PASS	FAIL	PASS	PASS	FAIL
<i>CumulativeSums</i>	PASS	PASS	FAIL	PASS	PASS	FAIL
<i>Runs</i>	PASS	PASS	FAIL	PASS	PASS	FAIL
<i>LongestRun</i>	PASS	PASS	FAIL	PASS	PASS	FAIL
<i>Rank</i>	PASS	PASS	PASS	PASS	PASS	FAIL
<i>FFT</i>	PASS	PASS	PASS	PASS	PASS	FAIL
<i>NonOverlappingTemplate</i>	PASS	PASS	FAIL	PASS	PASS	FAIL
<i>OverlappingTemplate</i>	PASS	PASS	PASS	PASS	PASS	FAIL
<i>Universal</i>	PASS	PASS	FAIL	PASS	PASS	FAIL
<i>ApproximateEntropy</i>	PASS	PASS	FAIL	PASS	PASS	FAIL
<i>RandomExcursions</i>	PASS	PASS	FAIL	PASS	PASS	FAIL
<i>RandomExcursionsVariant</i>	PASS	PASS	FAIL	PASS	PASS	FAIL
<i>Serial</i>	PASS	PASS	PASS	PASS	PASS	FAIL
<i>LinearComplexity</i>	PASS	PASS	PASS	PASS	PASS	FAIL

Table: NIST Test Results on TRNG Output for Different HPC Events on two different processors

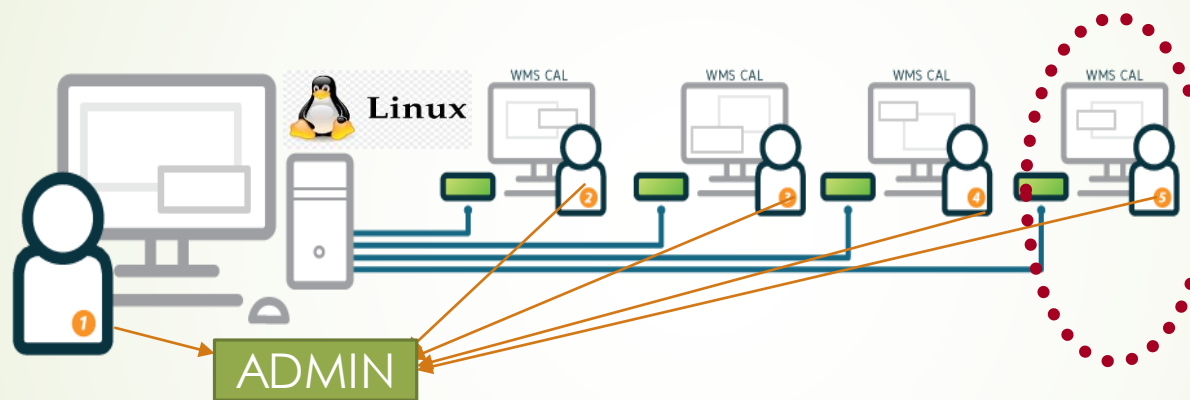
EXPERIMENTAL VALIDATION (Contd.)

AIS 20/31 Test	Intel		AMD	
	instructions	cpu-cycles	instructions	cpu-cycles
Procedure A				
T0	PASS	PASS	PASS	PASS
T1	PASS	PASS	PASS	PASS
T2	PASS	PASS	PASS	PASS
T3	PASS	PASS	PASS	PASS
T4	PASS	PASS	PASS	PASS
T5	PASS	PASS	PASS	PASS
Procedure B				
T6	PASS $d = 0.001990 < 0.025$ $s = 0.001080 < 0.02$	PASS $d = 0.001760 < 0.025$ $s = 0.000970 < 0.02$	PASS $d = 0.001640 < 0.025$ $s = 0.001120 < 0.02$	PASS $d = 0.001790 < 0.025$ $s = 0.000560 < 0.02$
T7	PASS $s_1 = 0.008000 < 15.13$ $s_2 = 0.050002 < 15.13$	PASS $s_1 = 0.079000 < 15.13$ $s_2 = 0.047869 < 15.13$	PASS $s_1 = 0.010000 < 15.13$ $s_2 = 0.049847 < 15.13$	PASS $s_1 = 0.047000 < 15.13$ $s_2 = 0.069748 < 15.13$
T8	PASS $s = 8.109696 > 7.976$	PASS $s = 10.479683 > 7.976$	PASS $s = 8.214734 > 7.976$	PASS $s = 9.975684 > 7.976$

Table: AIS 20/31 Test Results on TRNG Output for Different HPC Events on two different processors

EXPERIMENTAL VALIDATION (Contd.)

❖ Perturbation in TRNG Output in presence of an Adversary



Observation: Adversarial manipulation hampers the instruction counts but does not have any impact on the entropy of the least significant bits of the counter values

Reason: inherent chaos of a large number of concurrent process executions and optimization constructs of the Operating System and their effect on the underlying computer architecture modules

Attack scenario: An adversary running on the same processor core as the TRNG module can modify these HPC values in regular time intervals

EXPERIMENTAL VALIDATION (Contd.)

❖ Perturbation in TRNG Output in presence of an Adversary

NIST Test		AIS 20/31 Tests	
<i>Frequency</i>	PASS	Procedure A	
<i>BlockFrequency</i>	PASS	T0	PASS
<i>CumulativeSums</i>	PASS	T1	PASS
<i>Runs</i>	PASS	T2	PASS
<i>LongestRun</i>	PASS	T3	PASS
<i>Rank</i>	PASS	T4	PASS
<i>FFT</i>	PASS	T5	PASS
<i>NonOverlappingTemplate</i>	PASS	Procedure B	
<i>OverlappingTemplate</i>	PASS		PASS
<i>Universal</i>	PASS	T6	$d = 0.003479 < 0.025$ $s = 0.002547 < 0.02$
<i>ApproximateEntropy</i>	PASS		PASS
<i>RandomExcursions</i>	PASS		PASS
<i>RandomExcursionsVariant</i>	PASS	T7	$s_1 = 0.008429 < 15.13$ $s_2 = 0.094531 < 15.13$
<i>Serial</i>	PASS		PASS
<i>LinearComplexity</i>	PASS	T8	$s = 8.047369 > 7.976$

Table: NIST and AIS 20/31 Test results on TRNG Output for the HPC event instructions on Intel processor after adversarial modification

Hybrid Construction to Enhance Throughput

- Secured Hash implementation using Keccak algorithm
- Proposed design considers only the last 9 significant bits from the LSB at a periodic interval of 10ms
- Latency of 10ms of the generation of 9 random bits is inappropriate

Solution: Hybrid model which uses a shift register, the Keccak algorithm, and a control block by considering the random bits obtained from HPCs as input.

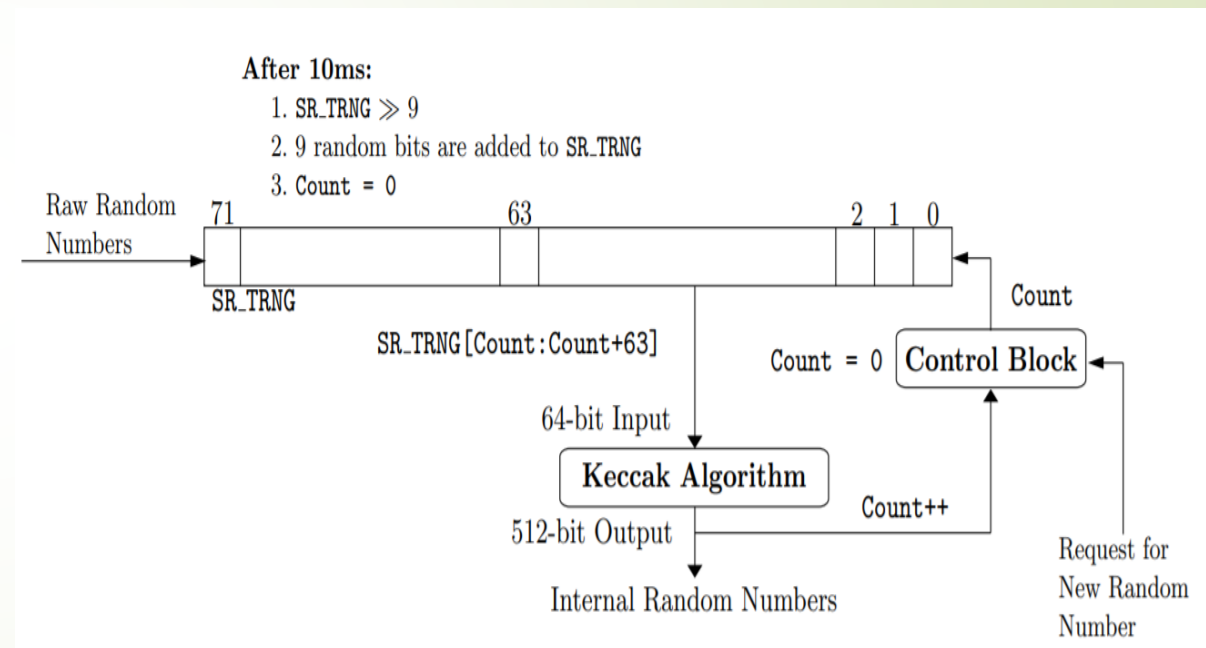


Fig: Hybrid Construction for generating internal random numbers

- Operational Modes: Initialization and Generation
- Maximum Throughput: 46,080 bits per second (or 45 Kbps)

Results for the Hybrid Construction

NIST Test		AIS 20/31 Tests	
<i>Frequency</i>	PASS	Procedure A	
<i>BlockFrequency</i>	PASS	T0	PASS
<i>CumulativeSums</i>	PASS	T1	PASS
<i>Runs</i>	PASS	T2	PASS
<i>LongestRun</i>	PASS	T3	PASS
<i>Rank</i>	PASS	T4	PASS
<i>FFT</i>	PASS	T5	PASS
<i>NonOverlappingTemplate</i>	PASS	Procedure B	
<i>OverlappingTemplate</i>	PASS		PASS
<i>Universal</i>	PASS	T6	$d = 0.004060 < 0.025$ $s = 0.005410 < 0.02$
<i>ApproximateEntropy</i>	PASS		PASS
<i>RandomExcursions</i>	PASS	T7	$s_1 = 0.499285 < 15.13$ $s_2 = 0.612501 < 15.13$
<i>RandomExcursions Variant</i>	PASS		PASS
<i>Serial</i>	PASS		PASS
<i>LinearComplexity</i>	PASS	T8	$s = 8.107012 > 7.976$

Table: NIST and AIS 20/31 Test results on TRNG Output for the HPC event instructions on Intel processor obtained from the hybrid construction

Comparing to Linux's RNG

- Linux based systems have special character file `/dev/urandom` providing an interface to the kernel's random number generator.
- However, several weaknesses of such random number generation is already reported in [Gutterman et. al.; S&P 2006].
- In order to stress the weakness, we collected random data using `/dev/urandom` and applied NIST Test suite on the output
- Thus our proposed approach can be used as a TRNG source in modern Linux based systems as an alternative to apparently weaker random number generator using `/dev/urandom`.

NIST Test	Intel	AMD
<i>Frequency</i>	FAIL	FAIL
<i>BlockFrequency</i>	FAIL	FAIL
<i>CumulativeSums</i>	FAIL	FAIL
<i>Runs</i>	FAIL	FAIL
<i>LongestRun</i>	FAIL	FAIL
<i>Rank</i>	FAIL	FAIL
<i>FFT</i>	FAIL	FAIL
<i>NonOverlappingTemplate</i>	FAIL	FAIL
<i>OverlappingTemplate</i>	FAIL	FAIL
<i>Universal</i>	FAIL	FAIL
<i>ApproximateEntropy</i>	FAIL	FAIL
<i>RandomExcursions</i>	FAIL	FAIL
<i>RandomExcursions Variant</i>	FAIL	FAIL
<i>Serial</i>	FAIL	FAIL
<i>LinearComplexity</i>	PASS	PASS

Table: NIST test results of Linux `/dev/urandom` on both Intel and AMD

Summary

- ✓ Components of architecture infuse a huge level of randomness because of Operating System optimization constructs and unpredictability of hardware interrupts.
- ✓ Hardware Performance Counters digitize the randomness of the architectural constructs and various experimental results using standard NIST, and AIS 20/31 Test suites show that these counters can indeed be considered as a TRNG source.
- ✓ Proposed TRNG construction is robust and fault tolerant in the presence of a powerful adversary
- ✓ Throughput Enhancement of the design is done by combining the TRNG module with Keccak hash implementation and a shift register to design a hybrid module which also qualifies NIST and AIS 20/31 Tests.

References

1. Chen, W., Che, W., Bi, Z., Wang, J., Yan, N., Tan, X., Wang, J., Min, H., Tan, J.: A 1.04 μ W truly random number generator for Gen2 RFID tag. In: 2009 IEEE Asian Solid-State Circuits Conference. pp. 117–120. IEEE (2009)
2. Cherkaoui, A., Fischer, V., Fesquet, L., Aubert, A.: A very high speed true random number generator with entropy assessment. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 179–196. Springer (2013)
3. Guñneysu, T.: True random number generation in block memories of reconfigurable devices. In: 2010 International Conference on Field-Programmable Technology. pp. 200–207. IEEE (2010)
4. Petrie, C.S., Connelly, J.A.: A noise-based IC random number generator for applications in cryptography. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications* 47(5), 615–621 (2000)
5. Robson, S., Leung, B., Gong, G.: Truly random number generator based on a ring oscillator utilizing last passage time. *IEEE Transactions on Circuits and Systems II: Express Briefs* 61(12), 937–941 (2014)
6. Yang, B., Rožić, V., Grujic, M., Mentens, N., Verbauwhede, I.: ES-TRNG: A Highthroughput, Low-area True Random Number Generator based on Edge Sampling. *IACR Transactions on Cryptographic Hardware and Embedded Systems* pp. 267–292 (2018)
7. Bayon, P., Bossuet, L., Aubert, A., Fischer, V., Poucheret, F., Robisson, B., Maurine, P.: Contactless electromagnetic active attack on ring oscillator based true random number generator. In: International Workshop on Constructive Side-Channel Analysis and Secure Design. pp. 151–166. Springer (2012)
8. Markettos, A.T., Moore, S.W.: The frequency injection attack on ring-oscillator based true random number generators. In: *Cryptographic Hardware and Embedded Systems-CHES 2009*, pp. 317–331. Springer (2009)
9. Rožić, V., Yang, B., Mentens, N., Verbauwhede, I.: Canary numbers: Design for light-weight online testability of true random number generators. In: *Cryptol. ePrint Arch., NIST RBG Workshop, Gaithersburg, MD, USA, Tech. Rep. vol. 386*, p. 2016 (2016)
10. Yang, B., Rožić, V., Mentens, N., Dehaene, W., Verbauwhede, I.: TOTAL: TRNG on-the-fly testing for attack detection using lightweight hardware. In: *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. pp. 127–132. EDA Consortium (2016)

Thank You!



This work was supported by the Defence Research and Development Organization (DRDO) through JCBCAT, Kolkata, India