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

Manaar Alam(✉), Astikey Singh, Sarani Bhattacharya, Kuheli Pratihar, and Debdeep Mukhopadhyay

Indian Institute of Technology Kharagpur, Kharagpur, India
alam.manaar@gmail.com, astikey070@gmail.com, tinni1989@gmail.com, its.kuheli96@gmail.com, debdeep.mukhopadhyay@gmail.com

Abstract. True Random Number Generators (TRNGs) are one of the most crucial components in the design and use of cryptographic protocols and communication. Predictability of such random numbers are catastrophic and can lead to the complete collapse of security, as all the mathematical proofs are based on the entropy of the source which generates these bit patterns. The randomness in the TRNGs is hugely attributed to the inherent noise of the system, which is often derived from hardware subsystems operating in an ambiguous manner. However, most of these solutions need an add-on device to provide these randomness sources, which can lead to not only latency issues but also can be a potential target of adversaries by probing such an interface. In this paper, we address to alleviate these issues by proposing an *in-situ* TRNG construction, which depends on the functioning of the underlying hardware architecture. These functions are observed via the Hardware Performance Counters (HPCs) and are shown to exhibit high-quality randomness in the least significant bit positions. We provide extensive experiments to research on the choice of the HPCs, and their ability to pass the standard NIST and AIS 20/31 Tests. We also analyze a possible scenario where an adversary tries to interfere with the HPC values and show its effect on the TRNG output with respect to the NIST and AIS 20/31 Tests. Additionally, to alleviate the delay caused for accessing the HPC events and increase the throughput of the random-source, we also propose a methodology to cascade the random numbers from the HPC values with a secured hash function.

Keywords: True Random Number Generator · Hardware Performance Counters · Cryptographic post-processing

1 Introduction

Random Numbers Generators (RNGs) form the backbone in the development of almost all of the devices requiring secure communication, device authentication, or data encryption. Applications of such devices include Smart Cards, RFID tags, and IoT devices. The unpredictability of random numbers is a crucial component in parameter and key generation for both symmetric and asymmetric cryptography, generation of random masks for padding data, generating session keys between communicating parties and several other secured applications. The RNGs rely on inherent chaos and unpredictability of various physical factors from the environment or hardware components to provide the much-needed randomness, which signifies that infinitely long history of previous random numbers provides no advantage over deciding or predicting the future random occurrences.

True Random Number Generators (TRNGs) derive its randomness from physical factors in the environment, and thus when implemented in hardware, the randomness is extracted from the physical parameters such as gate delay. Fault in design and fabrication procedure of TRNG results in insufficient entropy (randomness) or improper functioning of the TRNG. Depending on the type of randomness source and the post-processing involved, there have been quite a large number of TRNG designs in the literature. A popular TRNG based on Thermal Noise was first introduced by Intel [14] and recently can be found in [10,19]. The most commonly used entropy source for both FPGA and ASIC TRNG's is metastability, which can be found in [12,20]. Timing jitter in electronic systems was also considered as TRNG source in [11,26]. However, along with the design challenges involved in designing these TRNGs, an equally complex task is to develop test strategies for TRNGs. Owing to the fact that most tests are statistical and can only evaluate the statistical quality of the generated numbers and not their entropy, modern methods for certification of TRNGs involve a carefully chosen bag of tests. These tests mainly include: **(1)** *NIST* standard tests [22], which aims at estimating the min-entropy, i.e., evaluating the information content of the most likely outcome, and **(2)** *AIS 20/31* Tests [15] proposed by the German Federal Office for Information Security, which estimates Shannon entropy, targeted at evaluating the average information content of the random variable associated with the random source.

Furthermore, the TRNGs which are often implemented using external hardware are susceptible to physical attacks. In [16] attackers inject periodic signals to the power supply in order to destroy the randomness of the Ring Oscillators (ROs) by reducing the entropy of the generated keystream. In [7], the adversaries used strong magnetic fields to tamper the randomness generated by 50 ROs. In other instances, usage of power, clock glitches, and other techniques of physical attacks have threatened the deployment of TRNGs. Though there exist methods of on-the-fly testing of TRNGs which can be effective during such attacks [21,27], it would be desirable to develop TRNG sources which are available to a program without resorting to an external component. The TRNG should derive its randomness from the underlying hardware artifacts which are available in the computer architecture and which exhibit its randomness owing to the various processes which execute on them. Such an *in-situ* TRNG design would also make physical attacks more challenging, as compared to when the TRNG is an add-on hardware device which the adversary can target more effectively.

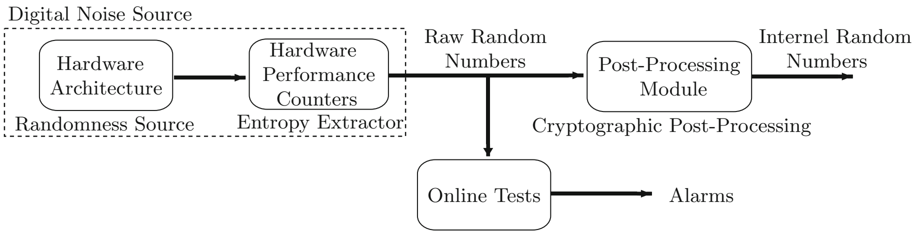

Fig. 1. Generic structure of the proposed True Random Number Generator (TRNG) circuit

In this paper, we analyze and propose an *in-situ* design for TRNG based on the randomness derived from various hardware activities as observed in the underlying computer architecture. These hardware activities (like instruction-counts, CPU-cycles, etc.) are observed through Hardware Performance Counters (HPCs) allowing detailed, low-level measurement of different process behavior executed in CPUs. HPCs provide valuable information about program execution, which were extensively used in the literature to attack strong cryptographic algorithms [3,9] and also used to detect the execution of malicious programs [1,2,5]. The information provided by HPCs are also used to identify vulnerabilities which aid reverse engineering of proprietary software [4]. However, modern processor vendors do not make any guarantee about determinism of these hardware activities (or HPC events) [25]. We analyze the source of non-determinism exhibited by these HPC events and aim to utilize this non-determinism in designing our TRNG module. We observe that the least significant bits of these HPC events display high-quality randomness and high entropy. We also propose a hybrid model coupled with a secure hash implementation in order to cope up with the latency in accessing the HPC events and thus to increase the throughput of the TRNG design where required.

Contribution

The major highlights of the paper in context to the general structure of a TRNG (as illustrated in Fig. 1) are explained in details as follows:

- We propose a TRNG derived from computer architecture, which thrives on the randomness observed through the architectural events from the underlying hardware. The HPC counters, which monitors these architectural events, exhibits inherent non-determinism in their implementation [25] and are also affected due to a vast number of processes running on a processor core in a fixed quantum of time. HPC event counters provide a cumulative count to these 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. The entropy reduces as we consider the bits more towards the Most Significant Bit (MSBs).

- These event counter statistics over the monitored application along with the background noise can only be observed at periodic intervals, which could create a bottleneck in terms of throughput. Thus, 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.

In the next section, we present a brief discussion on HPCs, which is the primitive tool to extract the randomness exhibited by the architectural component.

2 Preliminaries on Hardware Performance Counters

Hardware Performance Counters (HPCs) are special purpose registers present in most of the modern-day microprocessors which store the hardware related activities during the execution of a program. The HPCs provide detailed, low-level hardware utilization statistics to advance user modules and are thus very useful in code optimization and process tuning operations. The HPCs typically count the number of occurrences of various hardware events such as number of instructions executed, number of bus-cycles consumed, number of CPU-cycles consumed, different operations related to the cache memory, branch misprediction operations and many more events related to the architectural activities of a system. There are a wide variety of events that can be measured with HPCs, and the event availability varies considerably among CPUs and vendors. A full list of available events can be found in various vendor's architectural manuals. As these events are immensely useful in computer architectural design and optimizations, these were available to users having user-level privileges. However, various researchers developed security implications of such user-level accesses to these counter values for the cryptographic implementations, which led the security engineers to push the HPCs to higher privilege scale and thus the counters can only be accessed in modern systems with administrative privileges. It has already been shown in [25] that the HPC values obtained by monitoring a process are not deterministic in nature. In the next section, we analyze the reason behind the inherent non-determinism exhibited by the HPC events, which is the primary motivation behind the proposed TRNG design.

3 Non-determinism of HPCs and Motivation

All Linux based systems with kernel version 2.6.31 and above have a utility named `perf`, which can be used to access and read the HPC registers through the `perf_event` system call. The `perf` utility provides a simple command line interface to observe the detailed, low-level hardware based event counter values during the execution of a process. A user can monitor desired events for the entire duration of the execution or can observe them periodically with a fixed interval of time. The command to observe the values of a monitored event (`<event_name>`) with a fixed interval (`<interval_duration>`) of time during the execution of an executable (`<executable_name>`) is given as follows:

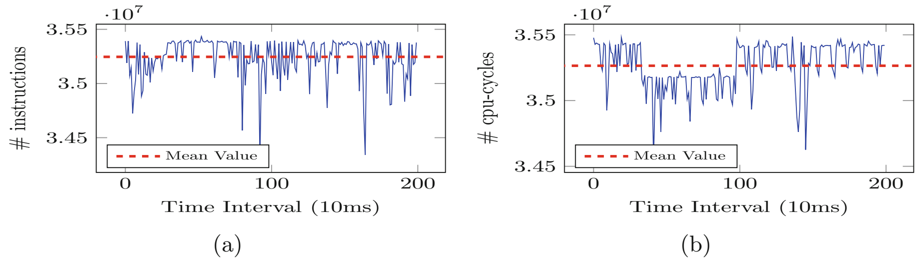

Fig. 2. Variation of the HPC events (a) `instructions` and (b) `cpu-cycles` monitored over an infinite loop on different time intervals

```
perf stat -e <event_name> -I <interval_duration> <executable_name>
```

In all of our experiments, we took a `C` code snippet which does nothing but loops infinitely, over which we observed various event counts such as `instructions`, `bus-cycles`, `cpu-cycles`, `cache-misses`, `branch-misses` and many more. As an example, we observed two performance counter events `instruction` and `cpu-cycles` over the executable of infinite loop with 10 ms[1] interval of time and the corresponding observations are shown in Fig. 2. In an ideal case, the HPC events `instruction` and `cpu-cycles` should report constant values over the duration of time, as the executable is doing nothing except looping infinitely. But, it can be observed from Fig. 2 that the number of instructions and the number of CPU cycles is not constant over time, which shows the significant amount of non-determinism exhibited by these performance counters.

Measuring exact event counts using HPCs can be difficult because of several external sources of variation such as program layout [18,24], multiprocessor variation [6], operating system interaction [17], measurement overhead [29], and hardware implementation details [23,24]. As discussed in [25], after carefully avoiding these sources of variation as much as possible, it was found that internal hardware interrupt is the potential source that leads to non-deterministic behaviour. Most of the HPC events get incremented an extra time for every hardware interrupt that occurs in the system. Hence, if an event is affected by hardware interrupts, then it cannot be a deterministic event, as it is impossible to predict in advance when these interrupts will happen. 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`, which also includes additional interrupt counts that happen outside of process context, adding extra assistance to the non-determinism of HPC events.

In order to validate this, we again monitored the events `instructions` and `cpu-cycles` as before and along with that we also measured the total number

[1] We selected 10 ms as it is the lowest interval of time that the `perf` tool supports, and thus corresponds to the highest supported frequency.

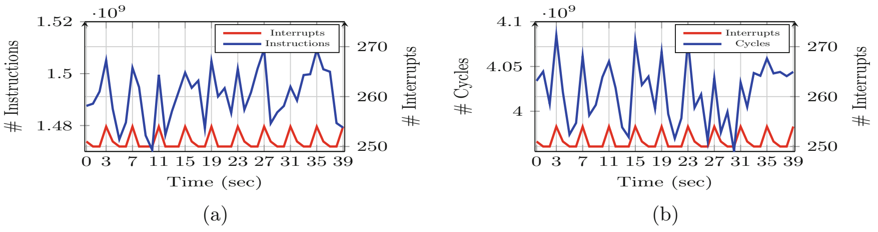

Fig. 3. Effect of hardware interrupts on the HPC events (a) `instructions` and (b) `cpu-cycles` monitored over an infinite loop on different time instances

of interrupts received per second using the Linux `mpstat` command which uses `/proc/interrupts` as a subroutine. We performed the experimentation on a per-core approach using the Linux `taskset` command. We present the effect of these hardware interrupts on both the HPC events in Fig. 3. We can easily observe from both the figures that whenever there is a surge in the number of interrupts, the counts of the events also increases validating the association between hardware interrupts and HPC events.

HPCs are registers dedicated to each core of a processor. Apart from being affected by the hardware interrupts, these event counts are also affected by the execution of various processes which are running in the background. The Operating System entirely administrates the execution of all the processes on the actual hardware of the architectural components, which are extremely complex to model as it is mostly dominated by the effect of speculative executions, out-of-order execution, interrupts, instruction prefetching and many more optimization techniques. Most of the modern processors are multicore, and an innumerable number of processes can get executed concurrently on each of these processor cores. The HPCs measure the event counts for all the events for all those processes which are concurrently running on the same processor core along with their fine-grained context switches. Thus if these HPCs are monitored for a per-core based approach, the counters are not only affected by the event counts from the operations in the monitored process but also from the all other processes which are running concurrently to the monitored application. Hence the background processes also have a significant impact on the event count of the target process, which is being monitored. In the next section, we propose an efficient construction of TRNG using the statistics obtained from these HPCs.

4 Randomness Extraction Using HPCs

In this section, we briefly discuss about the selection of appropriate architectural events to design the TRNG and the methodology to extract the randomness from such events. In the previous section, we have seen that the performance counter values range between a particular interval with low deviation from the mean, which makes the most significant bits of the observed values to be highly predictable. On the other hand, the noise component in these performance counter

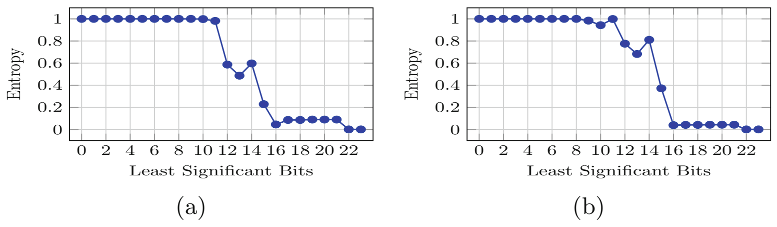

Fig. 4. Entropy of each LSBs for HPC event (a) `instructions`, and (b) `cpu-cycles`

values is very high if we consider only the least significant bits. The analysis of selecting the LSBs is elaborated next.

4.1 Selection of the Least Significant Bits

In the previous section, we claimed that the performance counter events which are observed over a straightforward executable exhibit a high source of entropy as they inherently possess a considerable amount of noise which is entirely contributed by the unpredictable hardware interrupts in the system and various processes running in the background. It should be further noted that the entropy of each bit position would not be the same for the binary sequences converted directly from the monitored values. The entropy is highest with the LSB, which can be considered as the most random bit output while the MSB is highly predictable. In support of our claim, we observed $500,000$ instances of the performance counter events `instructions` and `cpu-cycles`, and calculated the entropy for each bit position. Figure 4 shows entropy values of all the bit positions for both the events. We can observe from the figures that the LSBs have the highest entropy, and as we move towards the MSBs, the entropy gets reduced. Hence, instead of considering the observed HPC value, we transform the data into binary sequences and consider the last 9[2] bits for our further analysis.

4.2 Selection of HPC Events Using Yao's Next-Bit Test

In all of our experiments, we considered the sampling interval time as 10 ms. Hence, the `perf` tool will generate data points for any HPC event after each 10 ms time interval. Let us consider last n LSB bits of a single instance of data as the source of randomness. At any time instant t, we considered the sequence of bits $S(n,t) = \left(b_0^t, b_1^t, \cdots, b_{n-2}^t, b_{n-1}^t\right)$ derived directly from the value obtained by `perf` tool for each HPC events. Our conjecture to consider an HPC event as a source of TRNG is based on the idea that there will be no bias in predicting a bit, even if we know the previous values. In order to determine the bias in observation, we provide Yao's next-bit test [28], which we discuss as follows.

[2] 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.

Table 1. Next-bit test for different HPC events for $m = 4$

Known bits	Estimated value of $\hat{\mathbf{Pr}}[b_4^t = 0]$				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**

Table 2. Experimental setups for validation of the proposed claim

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

Suppose we know 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 $m < n$). Now, according to Yao's Next-bit test, we say that 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., $\mathbf{Pr}[b_m^t = 0] = 0.5 \pm \delta$), given the knowledge of $S(m,t)$, when δ is negligible (with respect to the security parameter). There are 2^m possibilities for $S(m,t)$, and we perform the test for all such possibilities. In order to estimate the $\mathbf{Pr}[b_m^t = 0]$, we consider N such sequences by observing an HPC event at N successive interval of time. Let the sequence $S(m,t)$ occurs at $\mathcal{T}$ times out of N possibilities. We now count the occurrences of $b_m^i = 0$ and $b_m^i = 1$ as $\mathcal{C}_0$ and $\mathcal{C}_1$ respectively, where $i = 1, 2, \cdots, \mathcal{T}$. We define the estimated probabilities as $\hat{\mathbf{Pr}}[b_m^t = 0] = \frac{\mathcal{C}_0}{\mathcal{T}}$ and $\hat{\mathbf{Pr}}[b_m^t = 1] = 1 - \hat{\mathbf{Pr}}[b_m^t = 0]$. Without loss of generality we first consider the case of $m = 4$, i.e., first 4 bits of the binary sequence is known. There are 2^4 possibilities of $S(m,t)$, which are shown in **Known Bits** column of Table 1. We observed $N = 500,000$ values for the events `instructions`, `cpu-cycles`, `cache-misses`, and `branches` and estimated the probability $\hat{\mathbf{Pr}}[b_4^t = 0]$ as discussed previously. The estimated probability values for all the events and for all the combination of **Known Bits** and the corresponding values of δ (as mentioned previously) are shown in Table 1. The first cell in Table 1 contains the value 0.499362, which signifies that for the event `instructions` if we known that the first 4 bits are 0000, then the estimated probability that the next bit will be 0 is 0.499362. We can also observe from the table that the corresponding value of δ is 0.000638. It is clear from the table that all the combinations have probabilities close to 0.5 for the events `instructions` and `cpu-cycles`. The corresponding values of δ are also negligible. In case of events `cache-misses` and `branches` the probability values are highly biased for some combinations with high value of δ and are shown with **bold** faces. The average values of δ for all the sequences corresponding to each of the performance counter events are also shown in Table 1 and justifiably these values are higher for the events `cache-misses` and `branches`. Similar results are observed for other values of m. Hence, we conclude after this analysis that the events `instructions` and `cpu-cycles` can act as better candidate for source of randomness while we discard the other two events `cache-misses` and `branches` in our further analysis. In the next section, we validate our conclusion through an extensive set of results with the help of NIST and AIS 20/31 Test suite.

5 Experimental Validation

In this section, we first provide results on TRNG output obtained from the raw HPC events followed by the results on the TRNG output in the presence of a strong adversarial perturbation.

5.1 Results on TRNG Output Obtained from HPC Events

All the experiments are conducted on two different processors as listed in Table 2, where the access to HPC events is available to users with administrative privilege. There exists a diverse set of HPC events which can be accessed via the `perf` utility. We considered some of the primitive events such as `instructions`, `cpu-cycles`, `bus-cycles`, `cache-misses`, `branches` etc., as the obtained values for each of these events are high compared to the other events. The perf statistics are recorded after every time interval of 10 ms for an executable which runs infinitely over time. The idea behind the selection of events which showed high values compared to the lower ones is because the ones reporting very high values can be expected to produce decent randomness in the Least Significant Bits. On the contrary, events which show low values as output are intuitively more predictable compared to the earlier case.

The NIST Test suite is observed to work the best for the events `instructions` and `cpu-cycles` in all the setups as mentioned in Table 2. The perf statistic is recorded for more than 15 hours of execution time, which resulted in altogether 10 sets, each set having more than 5.5×10^7 performance counter values. For each set, we selected the last 9 bits from the LSB of each observation and appended one after another to generate a consolidated binary string. We applied the NIST Test suite on this consolidated binary sequence. We furnish our results from the NIST suite for both Intel and AMD processors in Table 3. We can observe from the table that both the events `instructions` and `cpu-cycles` pass all the 15

Table 3. NIST test results on TRNG output for different HPC events on two different processors

NIST test	Intel			AMD		
	`instructions`	`cpu-cycles`	`cache-misses`	`instructions`	`cpu-cycles`	`cache-misses`
Frequency	PASS	PASS	FAIL	PASS	PASS	FAIL
BlockFrequency	PASS	PASS	FAIL	PASS	PASS	FAIL
CumulativeSums	PASS	PASS	FAIL	PASS	PASS	FAIL
Runs	PASS	PASS	FAIL	PASS	PASS	FAIL
LongestRun	PASS	PASS	FAIL	PASS	PASS	FAIL
Rank	PASS	PASS	PASS	PASS	PASS	FAIL
FFT	PASS	PASS	PASS	PASS	PASS	FAIL
NonOverlappingTemplate	PASS	PASS	FAIL	PASS	PASS	FAIL
OverlappingTemplate	PASS	PASS	PASS	PASS	PASS	FAIL
Universal	PASS	PASS	FAIL	PASS	PASS	FAIL
ApproximateEntropy	PASS	PASS	FAIL	PASS	PASS	FAIL
RandomExcursions	PASS	PASS	FAIL	PASS	PASS	FAIL
RandomExcursionsVariant	PASS	PASS	FAIL	PASS	PASS	FAIL
Serial	PASS	PASS	PASS	PASS	PASS	FAIL
LinearComplexity	PASS	PASS	PASS	PASS	PASS	FAIL

tests under the NIST Test suite. We perform the same experimentation on each of the 10 sets and obtain similar results for all the sets. Table 3 also shows the NIST Test results for the HPC event `cache-misses` on both the processors. We can observe that most of the tests under the NIST Test suite fails for this event, which aligns with the results shown in Table 1.

In order to further analyze the TRNG property of the events `instructions` and `cpu-cycles`, we applied the AIS 20/31 Test procedures on the consolidated binary output string as obtained before. The results of the tests for both Intel and AMD processors are shown in Table 4. We can observe from the table that all the tests under Procedure A and Procedure B of AIS 20/31 Test suite pass for both the events `instructions` and `cpu-cycles`. The details of the parameters mentioned under Procedure B can be found in the AIS 20/31 Test Manual. Hence, with the outcomes of these two test suites, we conclude that the HPC events which are affected by various hardware interrupt and the background noises can be effectively used to design a TRNG module.

5.2 Perturbation in TRNG Output in Presence of an Adversary

In the previous subsections, we tested the sources of entropy through normal process execution framework in a multi-core processor setup, where we show that the inherent chaos of the various process execution and the unpredictability of hardware interrupts have an extensive impact on the HPC values. We claim with suitable results that the values obtained from HPCs qualify for a pure computer architecture based TRNG. But we are also interested in understanding the effect on the HPCs in the presence of a powerful adversary.

Let us consider a server setup, where there are multiple users logged into the same server, and all of the users are having administrative privileges. Thus all of

Table 4. AIS 20/31 test results on TRNG output for different HPC events on two different processors

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 $s = 9.975684 > 7.976$

these users can observe perf statistics over executables which run on processor cores shared across various user processes. Hence it is feasible for an adversary running on the same processor core as the TRNG module to modify these HPC values in regular time intervals. We performed several experiments where the adversary process runs on the same processor core as the target core and uses `asynchronous perf ioctl system` calls to set the value of the HPC event `instructions` to zero periodically. This manipulation by the adversary hampers the instruction counts observed over a synchronous measurement procedure to a great extent. The range of the instruction counts varied widely when a concurrent adversary module refreshed the instruction counts, which is also expected if the adversary wishes to modify the counter values instead of resetting it. Any modification to the counter values by a powerful adversary does have an impact in changing the overall values of the instruction counts but does not have any impact on the entropy of the least significant bits of the counter values. The reason behind it is that of the 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. Hence, a powerful adversary needs to not only model the chaos exhibited by the background concurrent processes but also needs to have complete control of hardware interrupts appearing in the system, both of which is assumed to be a challenging task to execute. Without loss of generality, we tested the TRNG sequences generated by the HPC event `instructions` on the Intel processor in the presence of this adversary with both NIST and AIS 20/31 Test suites. The results are furnished in Table 5, which shows that all the tests under both of these test suites pass with the modified TRNG sequence. In the next section, we discuss a hybrid TRNG

Table 5. NIST and AIS 20/31 test results on TRNG output for the HPC event `instructions` on Intel processor after adversarial modification

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

construction using a secure hash implementation for enhancing the throughput of the design to cope up with the latency in accessing HPC events.

6 Hybrid Construction to Enhance Throughput

In this section, we describe an efficient generation of random bit string through a secured hash implementation using Keccak algorithm [8] followed by its validation as TRNG using NIST and AIS 20/31 Test suites. The design is simple yet effective in context to generating a high-speed sequence of random numbers. In the previous section, we elaborate on how True Random Numbers were obtained from the Hardware Performance Counter values. The proposed design only considers last 9 bits from the LSB of each cumulative sample of event count for a periodic interval of 10 ms. This latency of 10 ms of the generation of 9 random bits is inappropriate when compared to real-life random number generation requirements. Thus we bridge the gap with a hybrid model which uses a shift register, the Keccak algorithm, and a control block by considering the random bits obtained from HPCs as input. If an application asks for a random number within the interval of 10 ms, the hybrid model uses its deterministic algorithm to generate a more extensive number of random bits using the previous inputs.

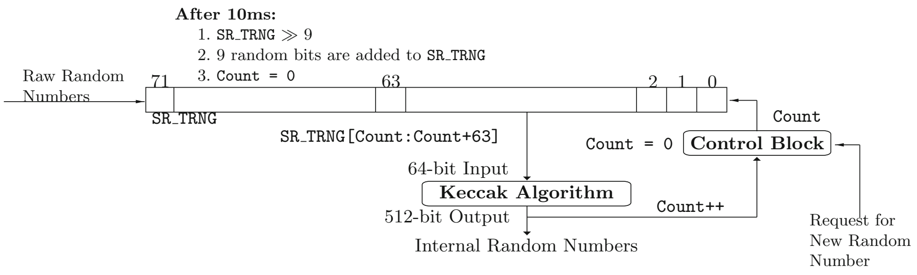

Fig. 5. Hybrid Construction for generating internal random numbers

6.1 Cryptographic Post-processing of the TRNG Output

The hybrid TRNG construction as shown in Fig. 5, takes the output of the true random number sequence obtained from the HPC events as its input and generates a sequence of more number of bits using a shift register (`SR_TRNG`), a Control Block, and Keccak (or SHA-3) Algorithm. The hybrid construction works with two operational modes as follows:

1. **Initialization:** The HPC based TRNG construction generates 9 random bits in every 10 ms interval. The hybrid construction waits for the first 80 ms after the start of the system. The shift register `SR_TRNG`, which is of length 72 bits, is filled from the LSB to MSB after each 10 ms in such a way that after first 80 ms the register `SR_TRNG` is filled with 72 bits of a random string. In this mode a register `Count` is also set to 0.
2. **Generation:** If a user needs a true random number from the system, it requests the Control Block to generate it. The Control Block then takes the 64-bit string `SR_TRNG[Count:Count+63]` and produces a 512-bit string using SHA-3 algorithm and provides these bits as an output to the request. The Control Block then increments the value of `Count` register by 1. After 10 ms the register `SR_TRNG` is right shifted by 9 bits, 9 random bits obtained from the new HPC value after the 10 ms are added to the shift register, and the register `Count` is again reset to 0.

Any user can obtain a maximum of $9 * 512 = 4608$ bits of a random string within the latency of 10 ms. Hence the maximum throughput of the hybrid design is $46,080$ bits per second (or 45 Kbps). It is evident that the throughput of the hybrid design is directly proportional to the length of the shift register `SR_TRNG`, which can be tuned to support different kinds of applications with varying requirements of throughput.

6.2 Results on TRNG Output Obtained from Hybrid Construction

The TRNG output obtained from the HPC event values are used as input to the hybrid construction. As discussed previously, after every 10 ms the shift register

Table 6. NIST and AIS 20/31 test results on TRNG output for the HPC event `instructions` on Intel processor obtained from the hybrid construction

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

Table 7. NIST test results on the output of Linux `/dev/urandom` on both Intel and AMD Processors

NIST test	Intel	AMD
Frequency	FAIL	FAIL
BlockFrequency	FAIL	FAIL
CumulativeSums	FAIL	FAIL
Runs	FAIL	FAIL
LongestRun	FAIL	FAIL
Rank	FAIL	FAIL
FFT	FAIL	FAIL
NonOverlappingTemplate	FAIL	FAIL
OverlappingTemplate	FAIL	FAIL
Universal	FAIL	FAIL
ApproximateEntropy	FAIL	FAIL
RandomExcursions	FAIL	FAIL
RandomExcursionsVariant	FAIL	FAIL
Serial	FAIL	FAIL
LinearComplexity	PASS	PASS

`SR_TRNG` holding the recent history of random bits from the TRNG is right shifted by 9 bits to accommodate fresh random bits. In an interval of 10 ms, we obtain the upper bound of 4608 bits of random binary string which requires only 72 bits of extra storage. The storage will be marginally higher for higher throughput design. We also take the output from the hybrid construction and run both the NIST and AIS 20/31 Tests on the sequences. Without loss of generality, results for the event `instructions` on the Intel processor are furnished in Table 6, which shows that the sequences pass all the tests under both the test suites.

7 Discussion

In this paper, we proposed a TRNG construction using the values obtained from the HPC events through the Linux based tool perf. However, all the Linux based systems have special character files `/dev/urandom` providing an interface to the kernel's random number generator, which gathers environmental noise from device drivers and other sources into an entropy pool. However, several weaknesses of such random number generation with a detailed cryptographic analysis is shown in [13]. In order to stress the weakness, we collected "random" data using `/dev/urandom` and applied NIST Test suite on the output. The result of the tests on both Intel and ARM processors are shown in Table 7. We can easily observe that apart from the *LinearComplexity* test under the NIST Test suite the dataset fails to qualify for all other tests. Since the dataset did not qualify the NIST Test suite, we did not provide any results on AIS 20/31 Test to show its weakness further. The objective of this discussion is to stress on the fact that the 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`.

8 Conclusion

In this paper, we showed that components of architecture infuse a huge level of randomness because of the Operating System optimization constructs and unpredictability of different hardware interrupts, which gets manifested through the Hardware Performance Counters. These 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. We have also shown that the proposed TRNG construction is robust and fault tolerant in the presence of a powerful adversary. The proposed TRNG module has a latency of 10 ms because of the time to access HPC events. Thus to enhance the throughput of the design, we combine the TRNG module with a simple yet effective Keccak hash implementation and a shift register to design a hybrid module which also qualifies NIST and AIS 20/31 Tests.

Acknowledgement. The authors thankfully acknowledge the Defence Research & Development Organisation (DRDO) for funding the project through JCBCAT, Kolkata, India.

References

1. Alam, M., Bhattacharya, S., Dutta, S., Sinha, S., Mukhopadhyay, D., Chattopadhyay, A.: RATAFIA: ransomware analysis using time and frequency informed autoencoders. In: 2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pp. 218–227 (2019)
2. Alam, M., Bhattacharya, S., Mukhopadhyay, D., Bhattacharya, S.: Performance counters to rescue: a machine learning based safeguard against micro-architectural side-channel-attacks. IACR Cryptology ePrint Archive 2017, 564 (2017)
3. Alam, M., Bhattacharya, S., Sinha, S., Rebeiro, C., Mukhopadhyay, D.: IPA: an instruction profiling-based micro-architectural side-channel attack on block ciphers. J. Hardw. Syst. Secur. **3**(1), 26–44 (2019)
4. Alam, M., Mukhopadhyay, D.: How secure are deep learning algorithms from side-channel based reverse engineering? In: Proceedings of the 56th Annual Design Automation Conference 2019, p. 226. ACM (2019)
5. Alam, M., Mukhopadhyay, D., Kadiyala, S.P., Lam, S.K., Srikanthan, T.: Side-channel assisted malware classifier with gradient descent correction for embedded platforms. In: PROOFS@ CHES, pp. 1–15 (2018)
6. Alameldeen, A.R., Wood, D.A.: Variability in architectural simulations of multi-threaded workloads. In: 2003 Proceedings of the Ninth International Symposium on High-Performance Computer Architecture, HPCA-9 2003, pp. 7–18. IEEE (2003)
7. Bayon, P., et al.: Contactless electromagnetic active attack on ring oscillator based true random number generator. In: Schindler, W., Huss, S.A. (eds.) COSADE 2012. LNCS, vol. 7275, pp. 151–166. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29912-4_12

8. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 313–314. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38348-9_19
9. Bhattacharya, S., Mukhopadhyay, D.: Who watches the watchmen?: utilizing performance monitors for compromising keys of RSA on Intel platforms. In: Güneysu, T., Handschuh, H. (eds.) CHES 2015. LNCS, vol. 9293, pp. 248–266. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48324-4_13
10. Chen, W., et al.: A 1.04 μW truly random number generator for Gen2 RFID tag. In: 2009 IEEE Asian Solid-State Circuits Conference, pp. 117–120. IEEE (2009)
11. Cherkaoui, A., Fischer, V., Fesquet, L., Aubert, A.: A very high speed true random number generator with entropy assessment. In: Bertoni, G., Coron, J.S. (eds.) CHES 2013. LNCS, vol. 8086, pp. 179–196. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40349-1_11
12. Güneysu, T.: True random number generation in block memories of reconfigurable devices. In: 2010 International Conference on Field-Programmable Technology, pp. 200–207. IEEE (2010)
13. Gutterman, Z., Pinkas, B., Reinman, T.: Analysis of the Linux random number generator. In: 2006 IEEE Symposium on Security and Privacy (S&P 2006), pp. 15–pp. IEEE (2006)
14. Jun, B., Kocher, P.: The Intel random number generator. White Paper, vol. 27, pp. 1–8. Cryptography Research Inc. (1999)
15. Killmann, W., Schindler, W.: A proposal for: functionality classes for random number generators. Ser. BDI, Bonn (2011)
16. Markettos, A.T., Moore, S.W.: The frequency injection attack on ring-oscillator-based true random number generators. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 317–331. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04138-9_23
17. Mc Guire, N., Okech, P., Schiesser, G.: Analysis of inherent randomness of the Linux kernel. In: Proceedings of the 11th Real-Time Linux Workshop. Citeseer (2009)
18. Mytkowicz, T., Diwan, A., Hauswirth, M., Sweeney, P.F.: Producing wrong data without doing anything obviously wrong!. ACM SIGARCH Comput. Archit. News **37**(1), 265–276 (2009)
19. Petrie, C.S., Connelly, J.A.: A noise-based IC random number generator for applications in cryptography. IEEE Trans. Circuits Syst. I: Fundam. Theory Appl. **47**(5), 615–621 (2000)
20. Robson, S., Leung, B., Gong, G.: Truly random number generator based on a ring oscillator utilizing last passage time. IEEE Trans. Circuits Syst. II Express Briefs **61**(12), 937–941 (2014)
21. Rožić, V., Yang, B., Mentens, N., Verbauwhede, I.: Canary numbers: design for light-weight online testability of true random number generators. In: NIST RBG Workshop, Gaithersburg, MD, USA, vol. 386, p. 2016 (2016). Cryptology ePrint Archive, Technical report
22. Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E.: A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, Booz-Allen and Hamilton Inc., Mclean, VA (2001)
23. Weaver, V.M.: Using dynamic binary instrumentation to create faster, validated, multi-core simulations. Ph.D. thesis, Cornell University (2010)
24. Weaver, V.M., McKee, S.A.: Can hardware performance counters be trusted? In: 2008 IEEE International Symposium on Workload Characterization, pp. 141–150. IEEE (2008)

25. Weaver, V.M., Terpstra, D., Moore, S.: Non-determinism and overcount on modern hardware performance counter implementations. In: 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 215–224. IEEE (2013)
26. Yang, B., Rožic, V., Grujic, M., Mentens, N., Verbauwhede, I.: ES-TRNG: a high-throughput, low-area true random number generator based on edge sampling. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**, 267–292 (2018)
27. 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)
28. Yao, A.C.: Theory and application of trapdoor functions. In: 23rd Annual Symposium on Foundations of Computer Science (SFCS 1982), pp. 80–91. IEEE (1982)
29. Zaparanuks, D., Jovic, M., Hauswirth, M.: Accuracy of performance counter measurements. In: 2009 IEEE International Symposium on Performance Analysis of Systems and Software, pp. 23–32. IEEE (2009)