Covert Channels through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations

Dmitry Evtyushkin
SUNY Binghamton
4400 Vestal Pkwy E
Binghamton, NY, USA
devtyushkin@cs.binghamton.edu

Dmitry Ponomarev
SUNY Binghamton
4400 Vestal Pkwy E
Binghamton, NY, USA
dima@cs.binghamton.edu

ABSTRACT

Covert channels present serious security threat because they allow secret communication between two malicious processes even if the system inhibits direct communication. We describe, implement and quantify a new covert channel through shared hardware random number generation (RNG) module that is available on modern processors. We demonstrate that a reliable, high-capacity and low-error covert channel can be created through the RNG module that works across CPU cores and across virtual machines. We quantify the capacity of the RNG channel under different settings and show that transmission rates in the range of 7-200 kbit/s can be achieved depending on a particular system used for transmission, assumptions, and the load level. Finally, we describe challenges in mitigating the RNG channel, and propose several mitigation approaches both in software and hardware.

CCS Concepts

•Security and privacy → Side-channel analysis and countermeasures; Security in hardware;

Keywords

Covert channels; Random number generator

1. INTRODUCTION

Modern computer systems are commonly shared among multiple groups of applications executing in different security domains. The security domain determines if an application can be granted access to certain data, perform privileged operations, or communicate with other applications. Such application isolation is typically implemented using system software, and the safety of user data critically depends on this support. The general principle of least privilege [46], which applies to many systems including the Android OS [15], advocates granting each application only a minimal set of permissions that are sufficient to support its proper functionality. For example, an application managing local personal data should be restricted from communicating over the network.

However, a sophisticated attacker or a malicious developer can use two colluding applications to create an attack that would send personal data over the network under such restrictions. The first malicious application has access to personal data and the second application has the network access. For consistency with previous works in this area, we refer to the first application as the *trojan* and the second application as the *spy*. To create an attack, the adversary first passes sensitive data from the trojan to the spy and then uses the spy to send the data over the network. However, since the trojan and the spy reside in two different security domains, a properly implemented permission system prevents them from directly communicating with each other. To bypass this restriction, the trojan and the spy can communicate using covert channels created by modulating the use of shared hardware resources in a microprocessor. Recent literature demonstrated many types of timing channels through shared CPU resources and their application to secret key reconstruction [36, 29], secret communication [17, 44] and bypassing of security mechanisms [13, 16, 25]. These implicit channels can even be used to compromise security of systems that provide hardware supported isolated execution environments [40, 10, 11, 57].

Covert channels are particularly dangerous in virtualized environments, such as computational clouds. Since virtualization naturally provides logical isolation between processes executing in different virtual machines, such environment seemingly provides a safe platform for manipulating secret data. Monitoring network traffic or direct information flows would prevent a malicious program from sending sensitive data to the outside world [6, 4, 42, 45]. Unfortunately, these safety guarantees can be bypassed if the attackers can communicate through covert channels.

In general, covert channels can be categorized into timing channels and storage channels [31]. Timing channels are created when the trojan performs manipulations with a shared resource in a way that interferes with the timing of some operations performed by the spy. In contrast, a storage channel is created by explicitly or implicitly writing a value to a shared resource by the trojan so that this write can be observed by the spy. Previous works demonstrated the exploitation of several shared hardware resources to create covert communication channels. These include the CPU functional units [53], the on-chip caches [56, 39], the AES hardware [22], the branch predictor [14, 12, 22] and the

CCS'16, October 24 - 28, 2016, Vienna, Austria
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4139-4/16/10…$15.00
DOI: http://dx.doi.org/10.1145/2976749.2978374

memory bus [55]. The temperature of the CPU cores has also been used as covert channel media and temperature-based channels have been shown to exist between air-gaped systems [18]. Researchers also demonstrated the feasibility of covert channels in GPGPUs [43].

While a number of covert channels inside a modern microprocessor chip have been demonstrated (as exemplified above), many of these channels have significant practical limitations. These include low transmission rate, difficulty of establishing and maintaining the channel, low resiliency to system noise and external interference, and fairly simple solutions to mitigate the channel. More details are presented in the related work section.

In this paper, we discover, implement and analyze a new covert channel that exploits hardware random number generation (RNG) module as the channel media. In most recent Intel processors (based on Skylake microarchitecture), the hardware RNG module is shared between all processor cores and is connected to them through the ring interconnect. The RNG module has a fixed number of precomputed random bits that are stored inside the module and are supplied to instructions that request a random seed. Once the random bits stored inside the RNG module are used, it takes a significant amount of time to regenerate them using the entropy source available in silicon. Consequently, the trojan can either exhaust the RNG module causing the spy to fail in its request for a random seed, or avoid using the RNG and allow the spy to succeed in its requests. Consequently, the trojan can use these two scenarios to either transmit a one or a zero to the spy.

Compared to previously demonstrated covert channels, the RNG channel has many important advantages from the attacker's standpoint. The RNG channel is fast, has low error rate, is easy to establish and maintain, and works reliably across CPU cores and virtual machines. The channel readings are easy to obtain without any system calls, because software (the spy in this case) is directly informed by the RNG module when an attempt to acquire a random seed fails. In this sense, the RNG channel is the storage channel and thus its maintenance does not require access to the processor timing infrastructure which often requires privileged access. The RNG module is rarely used in typical workloads, therefore the RNG channel is not impacted by the external interference. Finally, the RNG channel is difficult to mitigate in a non-virtualized system because the RNG instructions cannot be disabled by the operating system and are used directly from user space without system calls. In addition, techniques that disable or fuzz [21, 37, 51] with the processor timekeeping facilities will not provide protection in this case, because the RNG channel does not rely on timing infrastructure.

The main contributions and the key results of this paper are:

- We introduce a new covert communication channel that uses hardware random number generation module that is available in modern processors and is shared among all CPU cores.

- We demonstrate this channel on a recent Intel Skylake processor and show that the channel reliably works across CPU cores and virtual machines. Furthermore, the channel can be established and maintained purely from the user space without requiring any system calls.

- We quantify the capacity of the RNG channel under different scenarios. While the capacity of the hardware channel itself can be as high as 3 Mbit/s in idealistic scenario, we show that transmission rates between 7 kbit/s and 200 kbit/s can be realized depending on the system and the load level during transmission.

- We present a simple implementation of the RNG covert channel transferring bytes of data over the channel with the support for error correction and synchronization. This end-to-end channel supports transmission rates of up to 7 kbit/s.

- We explore the impact of other system activities (such as the intense GPU activity) on the channel quality and show that the impact is minimal, thus making the RNG channel robust to the internal system noise.

- We discuss the difficulties involved in mitigating the RNG channel and propose two software and two hardware-supported mitigation schemes. The software schemes involve modifications to the hypervisor to handle the timing of the `rdseed` instructions differently and also running a background thread to create the RNG noise. Hardware approaches include modifications to the RNG logic to remove the dependency of the `rdseed` instruction delays on the instructions generated by another thread.

2. BACKGROUND

Secure generation of truly random values is essential for producing encryption keys to support cryptographic operations. Using weak sources of randomness during key generation process is a well-known security threat [9, 32, 3, 28]. Random numbers can be generated either in software or in hardware. Software schemes [19] use various sources of entropy with non-deterministic nature, such as the disk seek time, the timing between user keystrokes, and the movement of the mouse. Software techniques are often slow, require user involvement, and do not have enough true entropy [26]. In contrast, hardware-based random number generators do not exhibit such limitations. Hardware generators rely on sources of non-determinism in silicon, such as the thermal noise [5], providing provable randomness at high speed [50].

To equip programmers with a fast, secure and easy source of random and pseudo-random numbers, hardware developers started to embed random number generators (RNG) inside the CPU chips. For example, Intel introduced the new hardware RNG [23] in their Ivy Bridge microarchitecture. In the initial offering, only pseudo-random number generator was available via the new `rdrand` instruction. The later Broadwell microarchitecture introduced the new `rdseed` instruction to add the capability to read true random numbers derived directly from the entropy source. AMD also added support for `rdrand` instruction and plans to add support for `rdseed` instruction in the upcoming microarchitecture. IBM POWER7+ also has hardware-based true random number generator [35].

Without loss of generality, the demonstration of the RNG covert channel in this paper is performed on Intel's CPU based on Skylake microarchitecture. Figure 1 depicts a high-level overview of the RNG mechanism in Skylake processor. The Entropy Source derives the entropy bits from thermal noise within the silicon at the rate of around 3 Gbit/s. The

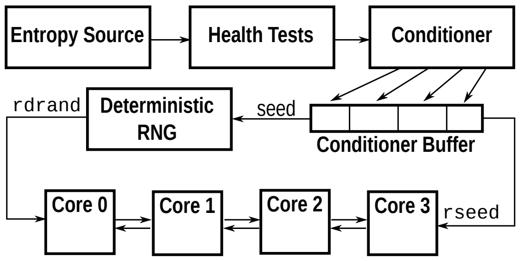

Figure 1: Organization of RNG Module in Intel Processors

quality of randomness is then verified by performing several health tests. Bits that passed the health tests form a pair of 256-bit numbers that are fed into the Conditioner. This circuitry distills the entropy into more concentrated single 256-bit sample. The 256 bits are stored as four 64-bit random values in Conditioner Buffer (CB), thus allowing to serve up to four `rdseed` instructions requiring 64-bit seeds. The outputs of the conditioner are then used directly by the `rdseed` instruction logic. In addition, the output bits of the conditioner are used to seed the deterministic random number generator. This generator uses CTR_DRBG pseudo random number generator and is accessed by the `rdrand` instruction.

While both of the above instructions are used to obtain random bits, there are significant differences between them. The `rdrand` instruction produces deterministic random numbers that depend on the instruction's previous outputs. Consequently, the `rdrand` instruction should not be used for security critical tasks requiring true randomness, such as the key generation. In contrast, the `rdseed` instruction outputs true random numbers. However, the `rdseed` instruction has a much lower throughput.

Based on our observations, while the throughput of the `rdrand` instruction is almost always sufficient to service all hardware threads constantly requesting pseudo-random bits, the resources of the `rdseed` instruction are easily exhaustible. *This temporal exhaustion of resources needed by the `rdseed` instruction is the key observation exploited in creating the RNG-based covert channel.*

Both `rdrand` and `rdseed` instructions explicitly inform the software on about successful completion by setting the carry flag. The `rdseed` instruction does not have any fairness mechanism built into it, therefore the availability of the random bits at the time of high demand has probabilistic nature.

Modern processors, such as the Intel's Skylake, are designed as complex System-on-Chip (SoC) with many components placed on the CPU die and interconnected together. These components include multiple cores, banks of the shared last-level cache, graphics processing unit (GPU), memory and PCI-express controllers and other peripheral hardware units. Recent Intel processors utilize the ring-based interconnect topology [27] in which all on-chip components are connected in a closed loop. The RNG module is organized as an independently clocked device built-in into the processor chip and connected via the ring. While publicly available documentation does not specify how exactly the RNG module is placed on this network, it is reasonable to assume that the connection is similar to any other device; this assumption is supported by the results of our experiments. Since the interconnection network is shared among all components, the activity of other devices can affect the timing of `rdseed` requests and result deliveries.

3. THREAT MODEL AND ASSUMPTIONS

We assume that the attacker controls two malicious applications in the system - the trojan and the spy. The trojan is a more privileged process that has access to sensitive data that it attempts to transmit to the spy process. No other communication channels (through the network, shared memory, file system, etc.) exist between the trojan and the spy, therefore these two processes can only communicate by means of covert channels. We also assume that both the trojan and the spy have access to shared hardware RNG module.

The system software is assumed to be secure, so that it properly enforces the access control and preserves legitimate information flows. The two processes only require normal user-level privileges. The RNG channel does not require access to performance counters, and therefore would work even if these counters are disabled as is commonly done on cloud systems [58]. Creating the channel also does not require access to processor timekeeping resources, therefore we assume that any protection that fuzz the resolution of the CPU timers can be in place.

4. COVERT CHANNEL THROUGH INTEL RNG HARDWARE

In this section we demonstrate how the RNG hardware can be utilized to construct a fast and reliable inter-core covert channel. We begin by demonstrating our ideas to implement covert communication within a single process, and then build up to practically usable channels.

4.1 RNG Channel in a Single-Process

The fundamental principle of data transmission through RNG is to control the contention for random values accumulated in the Conditioner Buffer (CB) by modulating the number of `rdseed` instructions executed at a given time interval. To transmit secret information, the trojan process either creates the high contention or the low contention for the use of CB resources. The high contention is created when the CB is exhausted and the new `rdseed` requests fail. In contrast, the low-contention condition is created when there are available entries in the CB. We assume that the high contention is associated with the trojan sending a value of "1" and the low contention is associated with the trojan sending a value of "0".

To estimate the theoretical maximum capacity of the RNG channel, we first describe its idealized implementation. The conditions required for this idealized scenario are, of course, impossible to support in a real-world environment. Our goal here is to estimate the upper bound on the transmission rate through the RNG channel. A CPU core can produce `rdseed` requests at a faster rate than the rate of refreshing the RNG hardware. Therefore, it is possible for the trojan process (or a group of processes) to create and sustain a situation where the CB has just enough bits to support a single `rdseed` instruction. At the same time, the spy process constantly executes one `rdseed` instruction and checks its status at the same rate as the rate of replenishing the CB. In such a setup, the trojan can send "1" by executing a single rdseed instruction and it can send "0" by not executing it. Hence, in order to estimate the time required to transmit a single bit from the trojan to the spy, we need to know the latency of the rdseed instruction and the rate at which the CB is replenished. This communication protocol's algorithm is demonstrated in the pseudo-code form in Listing 1.

To empirically determine the latency of the rdseed instruction, we developed a benchmark that executes and measures the latency of 2 000 rdseed instructions. To allow the RNG to refill the CB during each iteration, we executed 1 million nop instructions before executing the rdseed instructions. Executing this benchmark, we observed that the average latency of rdseed instructions was not identical when they were issued from different CPU cores. The results of this experiment are presented in Figure 2 with data collected on each core shown in a different color. As seen from the figure, the number of cycles taken by the rdseed instruction is lowest on core 0 (average of 402 cycles), and it is the highest on core 3 (417 cycles). The Simultaneous Multithreaded (SMT) virtual cores (cores 4, 5, 6 and 7) have similar latencies. Such variation in instruction latencies can be attributed to the specifics of the communication between the computing cores and the peripheral RNG module through the ring interconnect. For the CPU with 4 Ghz frequency, each rdseed instruction takes about 0.1 microsecond on the average.

The CB update rate was determined using the following experiment. We executed a stream of rdseed instructions on a core, checked their success status and counted the number of successful instructions. Since the core issues requests for random seeds at much higher frequency than the RNG hardware can produce, our experiment estimates the maximum throughput of the RNG and thus the refill rate of the CB. We observed that rdseed instructions can be successfully executed each 0.32 microseconds. Therefore, the CB is updated at a rate that is about three times slower than the rate at which the rdseed instructions can execute. Consequently, the CB update becomes the bottleneck in our idealized covert channel scenario, thus determining the maximum theoretical capacity of such idealized channel. Assuming that one bit can be transferred in 0.32 microseconds, the resulting bit rate (and thus the channel capacity) is 3.125 Mbit/s. This number represents the upper bound on the capacity of a channel that can be obtained through the RNG hardware.

4.2 Creating a Robust Channel in Single-Process Setting

A successful implementation of a covert channel requires solving two orthogonal problems. The first problem is how to transfer data through the shared media efficiently and with low error rate. The second problem is how to synchronize the spy and the trojan processes. Since our main goal is to study the methodology for using the RNG hardware as a covert channel and to estimate its capacity, we primarily focus on the first problem. We address the synchronization problem in Section 4.6. In addition, synchronization protocols presented in prior works [22, 48, 55] can be used to fine-tune the timings of communication phases.

The assumptions used in the previous subsection for estimating the maximum channel bit rate cannot be met in practice due to several reasons.

- The RNG hardware itself does not impose time limits in which the CB must be refilled. Although the hardware derives random bits at a constant rate, the built-in health checks can create some disparity and thus the CB can be replenished at arbitrary time.

- Other system-level activity such as context switches and accesses to shared caches by other processes can distort the perfect synchronization of the trojan and the spy.

- The RNG hardware is integrated into the processor chip as one of the peripheral devices sitting on a shared interconnect network. Therefore, the activity in this network can interfere with the fine-grain timing required for the idealized covert channel.

One option to bypass these limitations is to adapt the protocols used by the trojan and the spy by using slower bit rate or by using error-correcting codes [20]. However, such optimizations would significantly reduce the effective capacity of the channel, we describe the use of error-correcting codes later.

We now examine the interference problem due to the activity of other resources connected to the ring interconnect in more detail. Through experimentation, we observed that the Graphics Processing Unit (GPU) has the highest effect on the rdseed latency compared to other components connected to the ring interconnect. This can be an artifact of the RNG module placement on the ring, and high interference between the GPU-generated traffic and the RNG requests.

To assess the impact of such GPU interference, we measured the rdseed instruction latencies under two conditions: 1) the Graphical User Interface (GUI) is disabled and there is no GPU activity in the system; and 2) the system performs active 3D animation. Both parts of the experiment were conducted on the same core. The results are presented in Figure 3. As seen from the figure, the GPU activity introduces periodic, but significant delay into the RNG operation. The average values of the rdseed latency are 912 and 400 cycles with and without the GPU activity respectively. The slowdown is only observed during the phases of active animation. For example, GUI is enabled but the image is not moving, the results are similar to the case without GUI. Similar levels of interference were observed when the GPU was performing 2D animation (for example, a video playback) or computations on the GPU were performed when running OpenCL [49] code. On the other hand, our experiments showed that CPU-intensive activity does not increase the rdseed instruction latency.

Interference from the GPU makes it impossible for the trojan process to perform high-accuracy manipulations required to create the idealized channel described earlier. In particular, the variability of rdseed instruction timing introduces uncertainty in terms of when the request for random bits will be delivered to the RGN module. Thus, the trojan process can no longer keep the exhausted state of the CB, therefore making it difficult to construct a channel that exploits the fine-grained capabilities of the RNG.

As an alternative to fine-grain approach to channel creation, the trojan and the spy can use a more coarse-grained approach of treating the entire CB as a single unit and encoding transmission bits by altering high and low contention

Protocol 1 Fine-grained communication protocol

$D_{Send}[N], D_{Recv}[N]$: N bits to transmit and receive; N_{Prime} : Number of `rdseed` instructions to empty the CB
T_{rdseed} : Time needed to execute `rdseed` instruction; T_{Refill} : Time needed for RNG to refill **one** CB entry
T_{Prime} : Time needed to execute N_{Prime} `rdseed` instructions
wait(T): Wait for time T; `rdseed()`: Execute `rdseed`, returns **False** if failed

Trojan's operation:	**Spy's operation:**	Description:

for $i \leftarrow 0$ **to** $N_{Prime} - 1$ **do**
  `rdseed()`
	wait(T_{Prime})	Trojan prepares the CB by removing all of its entries

for $j \leftarrow 0$ **to** $N - 1$ **do**
  if $D_{Send}[i] = 0$ **then**
    wait(T_{Refill})
  else
    `rdseed()`
    wait($T_{Refill} - T_{rdseed}$)
	for $j \leftarrow 0$ **to** $N - 1$ **do**
	  wait(T_{rdseed})
	  $status \leftarrow$ `rdseed()`
	  if $status$ = **True then**
	    $D_{Recv}[i] \leftarrow 0$
	  else
	    $D_{Recv}[i] \leftarrow 1$
		Each cycle a single CB entry is generated. Trojan transfers 1 by consuming this entry, and 0 by allowing the spy to consume it

Listing 1: A fine-grained communication protocol (Protocol 1)

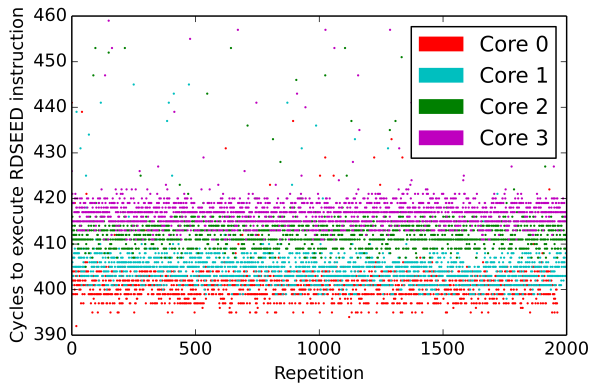

Figure 2: Latency of `rdseed` instruction on different CPU cores

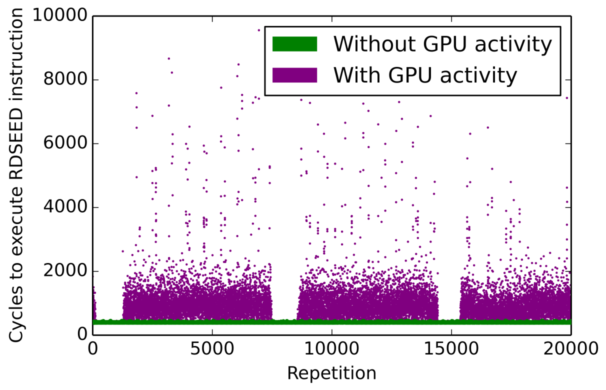

Figure 3: Latency of `rdseed` instruction with and without GPU interference

phases for the entire unit. To this end, we now describe a protocol that creates the RNG channel that is resilient to interference from the GPU and other hardware components connected to the ring. The noise resiliency comes with a slightly lowered channel capacity.

The pseudo-code of the coarse-grain RNG covert channel is presented in Listing 2. Transmission of a single bit of information through this channel consists of two stages. In *stage 1*, the trojan creates a desired contention state of the CB. If the trojan wishes to transmit a "0", it performs a busy wait. The waiting time (T_{Refill}) is the time needed for the CB to be refilled, as determined by the CB refill rate. If the trojan wishes to send a "1", it executes N_{Prime} `rdseed` instructions. To equalize the time needed to send both values prior to priming the CB, the trojan performs a short wait ①. The spy process remains inactive during *stage 1* and waits for this stage to finish. During *stage 2*, the spy probes the CB to determine the CB's state by executing N_{Probe} `rdseed` instructions and checking their status. If any of these `rdseed` instructions fails, then the spy detects the *exhausted* state of the CB, treating this event as the reception of "1". It is important to execute not one, but several `rdseed` instructions in order to mitigate possible CB refill during the probing process, as the exact timing of such refill is impossible to predict. The trojan remains inactive during *stage 2*.

The bit rate of this channel depends on the number of `rdseed` instructions required to exhaust the CB and to detect such exhaustion, and the number of cycles required for the CB to be fully refilled. Therefore, to estimate the channel bit rate, it is necessary to determine the values of N_{Prime}, N_{Probe} and T_{Refill}.

The best values for N_{Prime} and N_{Probe} depend on the capacity of the CB. Although the buffer itself is only 256-bit long, it is constantly updated. Therefore, the practical capacity (the number of bits the CPU can receive from the RNG without a failure) is higher than its size. We determined such capacity by conducting the following experiment. For each instruction sequence of size N, we performed several steps. First, we execute a large number (1 million) of

Protocol 2 Coarse-grained communication protocol

$D_{Send}[N], D_{Recv}[N]$: N bits to transmit and receive
N_{Prime} : Number of `rdseed` instructions to prime CB; N_{Probe} : Number of `rdseed` instructions to probe CB
T_{Prime} : Time for trojan to prime CB; T_{Probe} : Time for spy to probe CB; T_{Refill} : Time for a **full** CB refill
wait(T): Wait for time T; `rdseed()`: Execute `rdseed`, returns **False** if failed

Trojan's operation:	Spy's operation:	Description:
for $i \leftarrow 0$ **to** $N-1$ **do**	**for** $i \leftarrow 0$ **to** $N-1$ **do**	
if $D_{Send}[i] = 0$ **then** wait(T_{Refill}) **else** ① wait()($T_{Refill} - T_{Prime}$) **for** $j \leftarrow 0$ **to** $N_{Prime}-1$ **do** `rdseed()`	wait(T_{Refill})	Stage 1. Depending on the value to be sent, the trojan either removes all entries from the CB (primes it) or waits for the CB to refill
wait(T_{Probe})	$fail \leftarrow$ **False** **for** $j \leftarrow 0$ **to** $N_{Probe}-1$ **do** $status \leftarrow$ `rdseed()` **if** $status =$ **False then** $fail \leftarrow$ **True** **if** $fail =$ **True then** $D_{Recv}[i] \leftarrow 1$ **else** $D_{Recv}[i] \leftarrow 0$	Stage 2. The spy probes the CB by executing a sequence of `rdseed` instructions and checking their status

Trojan's operation:
```
for $i \leftarrow 0$ to $N-1$ do
    if $D_{Send}[i] = 0$ then
        wait($T_{Refill}$)
    else
①       wait()($T_{Refill} - T_{Prime}$)
        for $j \leftarrow 0$ to $N_{Prime}-1$ do
            rdseed()
    wait($T_{Probe}$)
```

Spy's operation:
```
for $i \leftarrow 0$ to $N-1$ do
    wait($T_{Refill}$)
    $fail \leftarrow$ False
    for $j \leftarrow 0$ to $N_{Probe}-1$ do
        $status \leftarrow$ rdseed()
        if $status =$ False then
            $fail \leftarrow$ True
    if $fail =$ True then
        $D_{Recv}[i] \leftarrow 1$
    else
        $D_{Recv}[i] \leftarrow 0$
```

Listing 2: A coarse-grained communication protocol (Protocol 2)

`nop` instructions allowing the CB to refill. The `nop` instructions serve the purpose of busy wait. Since no other processes are using the RNG hardware at this point, the buffer is guaranteed to be full by the end of this step. Following that, we executed the sequence of N `rdseed` instructions and checked for their status. If one of these instructions failed, we recorded a failure. If no failures were observed, we recorded a successful sequence.

We collected 1 million measurements for each sequence size and calculated average values. The results are presented in Figure 4. As seen from the figure, a process can always successfully execute a sequence of 5 `rdseed` instructions without any failures (failure rate 0%). Since each instruction fetches 64 bits, the practical capacity of the CB is 320 bits, which is 64 bits more than CB's physical size. A sequence of 6 `rdseed` instructions also has a low failure rate of 1.23%. As the number of instructions in the sequence continues to increase, the probability of success drops further since it is less likely for the RNG to refill the CB by the time the request arrives. The failure rates of sequences of size 7 and 8 are 2.86% and 15.98% respectively. Starting from sequence size of 9, the failure rate approaches 100%, with only a small number of successful sequences out of one million attempts.

The results of the above experiment can be used to detect the appropriate values for N_{Prime} and N_{Probe}. Selecting the value of N_{Prime} greater or equal to 5 ensures that the CB is exhausted during the prime stage. The value of N_{Probe} needs to be selected to allow the detection of missing CB entries with high probability, but with minimal false-positives. Choosing the value of N_{Probe} equal to 5 provides the maximum sequence with zero failure rate.

To select the optimal value of T_{Refill}, we performed another experiment. First, we primed the CB with a large number of requests for seed values, thus draining all random bits from it. Second, we allowed the RNG module to stay idle by executing a parameterizable number of `nop` instructions. Third, we repeatedly executed five `rdseed` instructions and checked the failure status of this sequence. For each value of the waiting period, we repeated the experiment one million times and calculated the average failure rate. The results are presented in Figure 5. As seen from the figure, any waiting period shorter than 3 000 cycles does not allow the CB enough time to be fully refilled to service five successive `rdseed` instructions. As the waiting time grows, the failure rate decreases. When the number

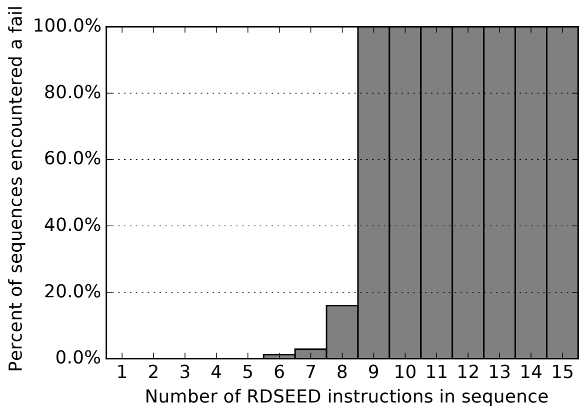

Figure 4: Failure rate of `rdseed` instruction groups executed on core 0

of cycles spent in waiting stage approaches 7 000, the failure rate becomes very low. Since the probabilistic nature of CB refill frequency makes it impossible to precisely compute the perfect value of T_{Refill}, one can experimentally select a value that results in a low waiting period and a low failure rate. We conservatively chose the waiting period value of 7 800 cycles for subsequent channel capacity estimation as it demonstrates a very low error rate.

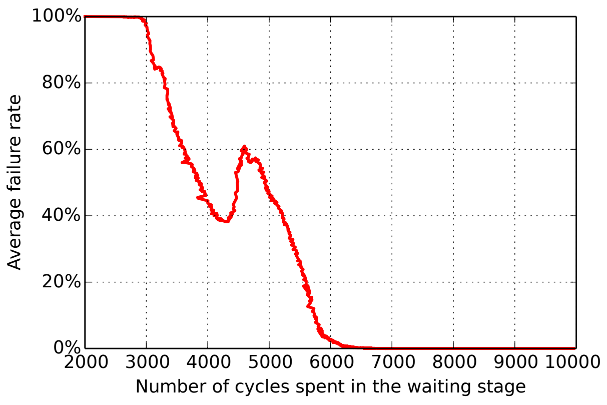

Figure 5: Percentage of failed 5-long `rdseed` instruction sequences after the CB is allowed to rest for a given number of cycles

Equipped with these parameters, we can now estimate the RNG channel bit rate under this setting. The bit rate can be calculated as $1/\big(Max(T_{Prime}, T_{Refill}) + T_{Probe}\big)$. In the Prime stage (stage 1), the trojan either executes five `rdseed` instructions (to transfer "1") or busy-waits (to transfer "0") for the length of T_{Refill}. Since each `rdseed` instruction takes about 400 cycles, the total time to execute five such instructions is 2 000 cycles. Therefore, the duration of the prime stage is determined by the larger value of busy-wait time and is 7 800 cycles. Note that when the trojan communicates a "1", it can first busy-wait for 5 800 cycles and then perform five `rdseed` instructions to match the time required to transmit a "0". Alternatively, the trojan can execute `rdseed` instruction for the whole duration of the 7 800 cycles prime period. The duration of the probe stage (stage 2) is 2,000 cycles, because only five `rdseed` instructions need to be executed by the spy. Therefore, 9 800 cycles are required to transmit a single bit of information $(2\,000 + 7\,800)$. Since our experimental processor is clocked at 4GHz, this translates into the bit rate of 408 kbit/s. Note that the channel bit rate is mostly dictated by the characteristics of the RNG module itself and is almost independent of the CPU speed. For example, in processors with lower frequency, the latency of `rdseed` instructions and the number of CPU cycles required to refill the CB will also decrease. Therefore, the channel bit rate is likely to stay at a similar level.

4.2.1 *Channel Capacity Estimation in Noisy Environments*

The RNG covert channel that we described above features high bit rate with a low error rate. However, several sources of interference exist that can increase the channel's error rate. An abstract scheme showing the channel operation under noise is shown in Figure 6. As we described previously, the GPU activity has a notable effect on the time needed by the CPU to service an `rdseed` instruction. When the GPU is performing active animation, the `rdseed` instructions are significantly slowed down, thus interfering with the data encoding mechanism described in Protocol 2. In particular, slowed down delivery of seed requests allows the RNG module to have more time to re-generate random bits and refill the CB. This distorts the correct channel functionality during the transmission of "1", because the correctness hinges on the predictable exhaustion of the CB. Based on our observations, the GPU activity can result in situations when even a CB that was fully exhausted during the priming stage will successfully provide five random seeds without a single failure. In such a case, the spy will incorrectly decode a "0". Erroneous switching from "0" to "1" is also possible. Such errors happen when the CB is not given a sufficient amount of time to re-generate the random bits, which results in a failure. It appears that the number of such errors does not depend on external interference from the GPU and has a probabilistic nature. Therefore, a longer wait time (T_{Refill}) results in less errors of this type, but the channel bit rate is reduced.

Using the parameters selected earlier, we constructed a benchmark to evaluate the error rate of the channel and calculate its capacity. We assume that our channel is memoryless, i.e. the output probability only depends on the current input, and not on previous channel states. To correctly compute the error rate under this assumption, we base our computations on transmitting random bits through the channel. Specifically, we generated 100 million random bits, transferred them through the RNG channel, and computed the error rate separately for transmitting zeroes and ones. We observed that the error rate was different when sending ones and when sending zeroes. Therefore, the channel can be characterized as a binary *asymmetric* channel with noise. The capacity of such channel can be calculated using the equation below [41].

$$C = \frac{\epsilon_0}{1-\epsilon_0-\epsilon_1} H_b(\epsilon_1) - \frac{1-\epsilon_1}{1-\epsilon_0-\epsilon_1} H_b(\epsilon_0) + \log_2\left(1 + 2^{\frac{H_b(\epsilon_0)-H_b(\epsilon_1)}{1-\epsilon_0-\epsilon_1}}\right) \quad (1)$$

where ϵ_0 is the probability of the trojan sending "0" and spy receiving "1" and ϵ_1 is the probability of a bit flip when the trojan sends "1". Such transitions are demonstrated in Figure 6. $H_b(p)$ is the binary entropy function of probability p which is defined as:

$$H_b(p) = -p\log_2 p - (1-p)\log_2(1-p) \quad (2)$$

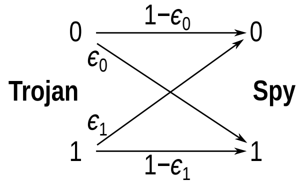

Figure 6: Binary asymmetric channel with noise between the trojan and the spy. Values ϵ_0 and ϵ_1 represent probabilities of errors when the trojan sends "0" and "1" respectively.

We evaluated the RNG channel based on Protocol 2 under the following settings.

- **No GUI**: System's GUI was disabled.
- **Static GUI**: The GUI was enabled, but no active animation was performed.
- **2D**: The GUI was enabled and a window was moved on the desktop while performing the benchmark.
- **3D**: A 3D benchmark was executed during the covert channel operation, creating a high GPU load.

The results of these experiments are presented in Table 1. As seen from the results, the GPU activity, even 3D animation, impacts the channel capacity only slightly, still allowing the attackers to maintain the bit rate of about 390 kbit/s.

	No GUI	**Static GUI**	**2D**	**3D**
ϵ_0	0.00066	0.19719%	0.20122%	0.19959%
ϵ_1	2.982e-05	0.00029	0.60933%	0.84875%
Bits per Channel Use	0.99579	0.98782	0.96276	0.95437
Channel Capacity (bit/s)	**406 443**	**403 198**	**392 965**	**389 540**

Table 1: Channel Characteristics Under Noisy Conditions

4.3 Detecting the Trojan and the Spy

Up until now we described the RNG channel in the framework of one process. The focus of previous discussions was on the hardware vulnerability itself and the quantification of a possible threat. If we now consider covert communication through the RNG module by two different processes, then a problem that is common to all covert channels arises: how to synchronize the trojan and the spy. To properly synchronize the transmission, both the trojan and the spy have to be made aware of the other's presence, so that the transmission and reception only occur when the two processes are running simultaneously. In addition, a mechanism for acknowledging the reception of data has to be incorporated. In this section, we design the synchronized communication protocol that utilizes the RNG hardware for sending both data and acknowledgments.

In realistic system operation, a process context switch (or a VM switch in virtualized systems) is the most significant obstacle to synchronized data transmission. For example, when the trojan process is scheduled by the OS, it has no information on whether the spy process is running on another core or not. The temporal inactivity of the spy process leads to the loss of large amount of bits — this is known as *burst erasure* [34]. In a similar spirit, it is difficult for the spy to distinguish the absence of the signal (when the trojan is switched out) from a sequence of zeroes sent by the trojan. While it is possible to adopt error correcting codes [30] that can correct burst erasures [34], this significantly complicates the design and lowers the channel capacity. In any case, the ability to detect the presence of the other party needs to be added to both the trojan and the spy. We now demonstrate how this can be accomplished using the same RNG hardware module that is used as the communication medium.

4.3.1 Detecting the Spy

As shown in the code of Protocol 2, the spy process measures the CB contention by executing five `rdseed` instructions at the probing stage of each communication cycle. Note that this activity by itself creates contention for the CB. Even if the CB was full prior to the probing stage, the five `rdseed` instructions executed by the spy will empty the CB. The next communication cycle begins right after the probing stage of the spy completes, with the trojan either waiting or priming the CB with new `rdssed` instructions. If the bit to be communicated by the trojan is "1", the trojan has the ability to not only execute the five priming instructions, but also check for their status. If the instruction sequence fails, the trojan can detect the presence of the spy since the CB was accessed right before the trojan began the priming stage. Such capability of the trojan to perform the CB priming and probing at the same time enables the detection of the spy's presence without incurring the loss of channel capacity. Therefore, the trojan can check for the spy's presence every time it primes the CB. When the trojan cannot detect the spy, it can temporary interrupt the transmission and continue probing the CB until the spy process is switched back in.

4.3.2 Detecting the Trojan

Detecting the trojan process is more tricky. The Naïve implementation of communication protocol relies on the Non-Return-to-Zero (NRZ) line code. In this code, "one" (the presence of contention) is encoded by a high voltage level, and "zero" (the absence of contention) is encoded by a low voltage level. The problem is that it is impossible for the spy to distinguish the absence of the trojan process from the trojan that sends a sequence of zeroes. We address this problem by adopting the Manchester coding. In Manchester coding, ones and zeroes are encoded as state transitions, rather than the states themselves. Therefore, during each communication cycle, even when the trojan sends "0", there will be the high and the low contention levels. This allows the spy to detect the absence of the trojan when it does not see high contention levels. While this approach lowers the bit rate of the channel, it allows for a clear and easy way for the spy to detect the absence of the trojan. Similar mechanism was used in [55] to synchronize their covert channel.

Combining the techniques for detecting the presence of spy and trojan processes, the RNG channel becomes self-synchronized. Occasional communication errors can be detected and corrected with proper error correcting codes.

4.4 Supporting Error Correction in Multi-Process Setting

In previous subsections, we estimated the capacity of the RNG covert channel in several different ways by modeling the behavior of the spy and the trojan in a single process scenario. This estimation quantifies the potential threat of exploiting the RNG hardware as a covert channel. In this section, we demonstrate the implementation of the RNG channel under realistic multi-process scenario where the spy and the trojan are two different processes running on the same machine. The implementation is based on the transmission principles described in the earlier sections, and we also take into account errors that can inevitably occur during the transmission.

As we described earlier, whether the `rdseed` instruction succeeds or fails is a probabilistic function even in cases when high demand for random seeds is created. Therefore, it is impossible to implement an error-free RNG channel using the existing RNG hardware. To implement reliable communication over the RNG channel in the presence of occasional errors, it is necessary to carefully select and use a proper Error Correcting Code (ECC). We now describe how this can be accomplished in a simple and effective way.

Most commonly used ECCs are block codes that operate on a block-by-block basis. A popular variation of such codes is the Reed-Solomon [54] (RS) group of codes. The RS code is capable of detecting and correcting a number of corrupted/erased symbols in a block. The number of symbols the code can correct depends on the specific parameters of the code and is highly configurable. An RS code is typically described by a set of parameters: s is the number of bits per code symbol; n is the number of symbols per block, and k is the message length in symbols per block. The parameter t represents the number of symbols that can be corrected in each block. It is defined as $t = \frac{(n-k)}{2}$. For example, the widely used code with $n = 255$ and $m = 223$ is usually denoted as RS(255,223) and it is capable of correcting up to 16 erroneous symbols. In order to select appropriate parameters of the code, one must determine the maximum possible number of corrupted symbols that can appear in a block and set the parameter k accordingly.

To make the RNG channel compatible with the RS codes, we adjusted the communication protocol based on Protocol 1 to send and receive bytes instead of the individual bits. We used Protocol 1 as a starting point for this experiment because it is based on simultaneous execution of the trojan and the spy and is easier to use in this case. A limitation of the byte-granularity channel is its lower tolerance to errors, because an error in any bit of a transmitted byte corrupts the entire byte. To overcome this limitation, we used the following approach. We reduced the speed of the trojan while keeping the speed of the spy at the maximum level. This allows the spy to perform multiple measurements (14 measurements in our setup) during the transmission of each bit by the trojan. This approach also has the additional advantage of easing the timing constraints.

Figure 7 demonstrates the spy's observations when the trojan transfers a byte with the bit value of 10110011. In this protocol, the spy makes the decision about the received value based on the majority of its readings. In most cases when the trojan transfers logical "0", the spy observes a sequence of all zeroes. Decoding of such readings is straightforward. However, when logical "1" is transferred, some of the readings will appear to the spy as zeroes. The spy can make a decision based on the number of observed ones in a sequence of readings. We adjust this threshold in such a way that minimizes the number of incorrectly decoded bits. For example, the leftmost bit in the sequence is decoded as logical one, because most of the readings are ones, as seen from the figure. The next bit is decoded as logical zero.

We set the number of `rdseed` instructions that the spy executes during each probing stage in such a way that allows the spy to monitor the state of the RNG module at any given time, but at the same time creating minimal parasitic contention from performing the probes.

We implemented this channel and discovered that it takes 200 657 cycles to transmit one byte of data, which involves making the required number of probes by the spy and storing the values read by these probes in memory. Thus, the attackers can transmit 19 934.42 bytes per second. This estimation assumes ideal inter-byte synchronization so that the primes and probes occur at exactly the same time. In practice, context switches and timing variations may distort this synchrony. We discuss possible solutions to synchronization and demonstrate one of them in Section 4.6.

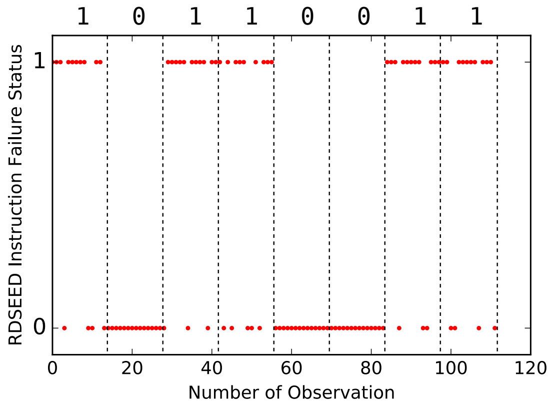

Figure 7: Spy's observation when the trojan transfers a byte of data over the channel.

Although the rate of about 20 kB/sec is lower than the theoretical rates that we showed above, the channel can still be considered as a high-bandwidth channel. We studied this channel under several noisy conditions. To be consistent with the previous measurements, we evaluated the channel under the same effect of the GPU noise as we did in Section 4.2.1. We did not measure the impact of legitimate programs that use RNG hardware in our evaluation model, because normal programs request random seeds very rarely, usually only during startup and initial key generation. Such events do not create a steady contention for the random seeds and thus do not introduce significant noise. Programs that have high demand for the random values (e.g. Monte Carlo simulations) rely on pseudo-random [8] numbers, rather than on true random seeds. This is one of the critical advantages for considering the RNG hardware unit as covert communication medium.

To perform this experiment, we first set up the RNG channel equipped with ECC to transfer blocks of 255 bytes. Following that, we evaluated the channel under different noise conditions and recorded the maximum number of errors observed in any of the transferred blocks for each environment setting. This statistics allowed us to select the best value of parameter k for each case. This parameter was selected to allow the correction of all errors that occurred during our experiments. The resulting error rates, along with the value of parameter k and the resulting channel bit rates are presented in Table 2. As seen from the presented data, the observed error rate increases with the interference from the GPU which confirms the earlier observed behavior. However, we note that covert channels are considered more dangerous in server machines which usually do not have graphics. Even in the case of active 3D animation performed in the background, the channel error rate is still within acceptable ranges and can be easily corrected with the use of ECC.

Finally, we note that the bit rates reported here should

	No GUI	**Static GUI**	**2D**	**3D**
Maximum block error rate ($= t$)	6	11	33	40
k	243	233	189	175
Bit rate with ECC (bytes/s)	18 996	18 215	14 775	13 680

Table 2: Channel Characteristics Under Noisy Conditions

not be interpreted as the maximum achievable practical bit rate. Further optimizations such as more sophisticated encoding/decoding and aggressive timing optimizations can achieve higher rates. Our goal was to demonstrate that reasonably high transmission rates can be achieved even with a simple implementation.

4.5 Capacity Estimation of Multi-process Channels

With the trojan and the spy detection capability in place, the actual channel capacity when operating in a multi-process setting will be limited by the amount of time that the spy and the trojan execute simultaneously on two different cores. Naturally, this time depends on the OS scheduling and also on the external system load (e.g. on other processes also running in the system at the same time). To estimate the impact of scheduling and interference on the achievable transmission rate, we performed the experiments on three different systems: a desktop system with 4-core processor running standard user environment applications, a server-class system with 16 computing cores, and a dedicated virtual machine running on Amazon EC2 cloud. For each experiment, we measured the percentage of trojan's cycles during which the spy also executes on one of the other cores. We did not change the scheduler policy and allowed normal competition for the time resources with other processes. The results are presented in Table 3.

	Desktop		Server		EC2
	Clean	Loaded	Clean	Loaded	Normal Load
P	0.937	0.319	0.991	0.179	0.850
Protocol 2 bit rate (kbit/s)	187.4	63.8	198.2	35.8	170
ECC channel bit rate (kbit/s)	71.212	24.244	75.316	13.604	64.6

Table 3: Fraction of time **P** the trojan runs simultaneously with the spy and estimated bit rate

For each system, the table shows the percentage of cycles (out of the total trojan's cycles) when the trojan and the spy are co-executing together (**P**). It also shows the resulting channel capacity that would be achieved on each system under each scheduling scenario and the additional load from external processes. We used two baseline capacities. First is the upper bound capacity of 400 kbit/s as shown in Table 1 (Protocol 2 channel) and the second capacity of 152 kbit/s (18.9 kB/S) with built in error correction as shown in Table 2 (ECC channel). We used the value of **P** to adjust the capacities under the normal scheduling policies. For example, if the value of **P** is 0.5, then 50% of the time the trojan executes together with the spy and the capacity in this case would decrease by two times.

For the desktop system in a clean state (no other processes are running), the spy runs alongside trojan 94% of the cycles, resulting in the channel capacity of 375 and 142 kbit/s for the channel based on Protocol 2 and the ECC-equipped channel respectively. In order to allow the spy to detect when the trojan is not executing, the usage of Manchester codes is needed as we discussed in Section 4.3.2. Incorporating Manchester codes into the ECC channel results in the reduction of the throughput by two times. Although 8 bits are still transferred and decoded, only 4 bits are then extracted by the spy. However, to be consistent with our previous ECC parameters, we let the ECC still operate on the symbol size s of 8. The spy simply combines two 4-bit symbols into one and uses it in error correction. As a result, the capacity is lowered by a factor of two. Resulting bit rates when Manchester codes are used are 187 and 71 kbit/s for Protocol 2 and ECC channel respectively. We also experimented with noisy environment, where two CPU-bound external processes are scheduled on the cores where the trojan and the spy execute. In this case, the co-scheduled time is only about 32%, resulting in the channel capacity of 64 and 24 kbit/s for the two channels (again, accounting for the effects of Manchester encoding). On the server, the co-scheduled percentages are 99% which results in bit rates of 198 and 75 kbit/s. In noisy environment, the two processes are co-scheduled 18% of the time, resulting in the bit rate of 36 and 14 kbit/s for both channels. We also performed scheduling experiments on the EC2 cloud with normal load. In this setting, the experimental VM shares the hardware with other VMs running on the cloud. The trojan and the spy run together 85% of the time, resulting in the channel capacities of 170 and 65 kbit/s for the two channels. In all of these scenarios, a high-capacity channel can be realized.

4.6 Channel Synchronization: Simultaneous Scheduling Intervals

In previous sections, we demonstrated the RNG covert channel in single and multi-process settings, discussed how to detect the presence of trojan and spy, and demonstrated how to adapt the transmission protocol to incorporate the error correction using Reed-Solomon ECC. However, previous discussions and results assume ideal internal synchronization and the described channels are suitable for the transmission of short byte-sized messages. We conclude the attack part of this paper by demonstrating a simple covert channel synchronization mechanism to support the transmission of longer messages. Our implementation is simple and is only intended to show practical end-to-end RNG channel realization. Higher transmission rates can be realized using more complicated fine-grain synchronization schemes, this is left for future work.

To understand the need for synchronization, consider a situation when the trojan is interrupted by a context switch in the middle of its message, and later resumes execution. In this case, both the trojan and the spy need to know that the transmission resumes when the trojan is rescheduled. More generally, both processes need to know when the message begins and when it ends. Synchronization of this nature is not a problem that is unique to the RNG channel — any covert channel needs to be synchronized. Several previous studies considered this problem. For example, [22] used the idea of a pilot signal. For simplicity, we implemented a different approach that we call Simultaneous Scheduling Intervals (SSI). The key idea is to schedule the trojan and the spy for short time intervals, so that they run simultaneously multiple times per second without interrupts. During each interval, the transmission of a single byte occurs. The synchrony is achieved by accurately aligning the times when the trojan and the spy wake up for the next interval.

To implement SSI, we relied on the `timer_create` POSIX interface available on Unix-like systems. Specifically, both the trojan and the spy use this system call to request to be woken up 1,000 times per second. During each interval, the two processes send and receive a small amount of data. Due to the very short duration of the execution periods, the trojan and the spy are not interrupted by the OS. To synchronize the clocks more closely, both processes perform some additional tuning during the first interval. Specifically, they read the system timer (with microsecond resolution), and insert the additional sleep time to the first execution interval to end it at the millisecond boundary, or as close to it as possible. After that, the scheduling of the consequent intervals occurs at the millisecond boundaries, with slight deviations due to system noise. The OS assures that these deviations do not accumulate over consecutive intervals and synchronous execution of the two processes is sustained. Our experiments showed that these deviations do not impact the transmission and error-free channel can be realized in this manner. Note that the nature of this particular approach to synchronization also makes the spy and the trojan detection unnecessary, thus obviating the need for Manchester coding. However, Manchester coding will still be needed if more fine-grain and high-performance synchronization schemes are used.

The simplest way to use the SSI synchronization is to transmit a single bit during each interval. This results in a bit rate of 1 kbit/s, but does not require ECC or Manchester coding. Our experiments showed that on a clean system such bit rate can be reliably achieved with no errors, as the clocks of the trojan and the spy are very closely synchronized with typical deviation of about 3-5 microseconds. It is also possible to use SSI for transmitting a byte of data per interval. However, occasional decoding errors occur in this case, requiring ECC for error correction. According to our experiments, the average error rate of 0.9% was observed. The maximum number of errors in a block of 255 bytes (ECC parameter t) was observed to be 19. Therefore, for this experiment 38 ECC bytes of ECC are needed inside every 255 bytes of transmitted data to correct all errors. Consequently, the channel bit rate is 6.8 kbit/s. While representing a true end-to-end transmission rate that we achieved on a real system, this number should not be viewed as the best achievable rate. Timing optimizations and more efficient fine-grain synchronization can increase the channel capacity further.

5. MITIGATING THE RNG COVERT CHANNEL

The new covert channel through the RNG hardware module presented in this paper imposes security risks that have to be addressed. In general, several classes of mitigation techniques against side channels and covert channels to eliminate information leakage are well known from prior literature, and some of these can be adapted to mitigate the RNG channel. These solution classes include static partitioning of hardware resources, approaches targeted at equalizing access delays by different instructions, and approaches designed to complicate or disable the usage of timekeeping infrastructure by the attackers. In terms of closing the RNG covert channel, the defense goal is to ensure that the latency (and, more specifically, the failures) experienced by an `rdseed` instruction executed by one process is not impacted by any activity of another process.

Compared to many other architectural covert channels, the RNG channel is relatively more difficult to mitigate. This is because the communicating processes in most covert channels have to rely on a measurement mechanism by either using hardware performance-monitoring counters [22] or the processor time stamp counter [39]. The access to these counters is typically offered through the use of specific instructions, such as `rdpmc` and `rdtsc` on x86 systems. However, since these instructions can carry a potential security risk, the hardware designers make their availability configurable. In particular, system software can set up these instructions to be either available to software executing at any privilege level, or restrict it to only software executing in ring 0. Restricting the usage of these instructions to ring 0 programs essentially closes the user-level covert channels that rely on such measurement.

Since accurate and frequent time measurement is critical for proper functionality of many applications, techniques have been proposed to make the timing measurements less precise (thus distorting the timing channels), while maintaining the capabilities of benign applications [37, 51]. However, fuzzing with or disabling the time measurement infrastructure does not close the RNG channel, because the CPU explicitly informs the software about the availability of random bits. Moreover, operating systems in non-virtualized environments cannot restrict programs from executing the `rdseed` instructions, thus making it impossible to mitigate the RNG channel by simply applying configuration tweaks in such systems.

In the rest of this section we propose two software mitigation techniques that can either disable the RNG channel or bring its capacity down to impractical levels. In addition, we propose two hardware-supported mechanisms to make the design of RNG modules in future processors immune to the exploitation as covert channel media.

5.1 Software Mitigations

First, we describe possible software-only protections from the RNG channel. We propose two software-based mitigations: one approach is based on the support available in Intel-based virtualized systems, and the other approach relies on executing a background thread to add constant load on the RNG hardware.

5.1.1 Virtualization Based Solution

As we discussed above, the OS does not have the capabilities to directly disable the RNG channel. However, it is possible to mitigate the RNG channel in a virtualized system. Specifically, the Intel virtualization technology [24] (VMX) allows to configure the hardware to cause a VM exit operation and trap into the Virtual Machine Monitor (VMM) every time `rdrand` and `rdseed` instructions are executed by a guest VM. This provides an effective mechanism to control accesses by guest VMs to the RNG hardware and allows for a number of defenses to be implemented within the VMM to disable the RNG covert channel.

One mitigation technique that can be implemented within the VMM is to completely emulate the functionality of the RNG hardware in software. Specifically, whenever a trap to the VMM occurs after the guest VM executes the `rdseed` instruction, the VMM provides a software-generated random value. Unfortunately, this approach loses the benefits of pure hardware-based random seed generation.

Another solution is to add the capability of distributing random seeds to the VMM itself. In particular, the VMM software can execute the `rdseed` instructions to fill up its own pool of random seeds and then pass these seeds to the guest VMs whenever a VM makes an attempt to execute the `rdseed` instruction.

Finally, the VMM can introduce a delay following a trap from the VM upon encountering the `rdseed` instruction and then return the control to the VM. If the delay is sufficiently long to allow the RNG hardware to completely replenish its random bits, then no `rdseed` instruction will experience failure because they are separated by a sufficient number of cycles.

The VMM-based mitigations described above will currently only work on Intel processors. While AMD processors do not implement the `rdseed` instruction, they support the `rdrand` instruction. The support for `rdseed` is expected to be added to AMD processors in the next microarchitecture revision. According to recent documentation [1], the AMD virtualization hardware does not allow configuring it to produce a VM exit when executing `rdrand`. Unless this support is added with the introduction of the `rdseed` instruction, alternative mitigation strategies for AMD processors have to be used.

5.1.2 *Equalizing the RNG Loading*

If the VMM-based solution described above can be used, it represents the simplest way to mitigate the RNG covert channel. However, virtualization is not always used or is not always available. Another potential limitation of VMM-based approach is that it may not always be possible to modify the VMM's code. In those cases, alternative solutions need to be used.

Another approach to mitigating the RNG channel is to make the load on the RNG module equal at any time, thus making it impossible for the attackers to encode information using the RNG state. For example, if we introduce constant high-intensity demand for the RNG services in the background, then the spy process will always detect the high-contention RNG state, making it impossible to distinguish between one and zero for data transmission using the RNG channel.

We propose to achieve such constant pressure on the RNG hardware by dedicating a hardware thread to repeatedly execute the `rdseed` instructions at a high rate. To evaluate the effectiveness of such mitigation, we repeated the experiment described in Section 4.2.1 by transmitting a large number of bits (100 million bits) through the RNG channel using Protocol 2. The only difference was that this time we also executed the protection thread in the background (on one of the available cores) that presented constant high-level demand for the RNG hardware. As expected, the noise process significantly interferes with the functionality of the RNG channel. Specifically, almost all of the transferred bits were decoded as "1" by the spy process, regardless on their actual value. For example, of the 50 million zeroes that were transmitted, only 556 were correctly received by the spy. This translates into the values of ϵ_0 and ϵ_1 of 0.99998886 and 2.54e-06, the transmitted bits per one channel use of 2.132e-06, and the resulting theoretical bit rate of 0.87 bit/s. An extremely high error rate caused by this protection makes the channel with such a low bit rate (less than a bit per second) unusable for transmitting any practical information.

While the protection based on running a spurious thread looks expensive, it can be enabled only when the system detects that potential trojan and spy processes have been scheduled. This protection also does not impede the capability of processes to obtain small random seeds. Based on our experiments, a process can still receive seeds, possibly after repeating the attempts several times. In particular, when a single RNG-equalizing process is present, a legitimate process needs to perform 5 failed `rdseed` attempts on average before succeeding. Since random seed generation is an infrequent operation in typical scenarios during normal program execution, the normal system functionality is not significantly distorted by this mechanism.

5.2 Hardware-supported Mitigations

In future systems, the hardware design itself can be modified to mitigate the RNG-based covert channel. We describe two possible approaches that future RNG modules can implement to support security: equalization-based approach and partitioning-based approach.

5.2.1 *Equalizing Delay of* `rdseed` *Instructions*

To eliminate the RNG-based channel, future RNG hardware can be designed in a way that ensures uniform latency of every `rdseed` operation and eliminates random seed request failures. First, the explicit failure signal that the RNG hardware currently uses to inform software about the lack of random bits can be eliminated. Instead, the `rdseed` instructions can be required to wait until the random bits are replenished if the module is currently out of bits. This tweak does not completely eliminate the vulnerability, but requires the attackers to use timing infrastructure instead of capitalizing on the explicit failure signal. This makes the attack more difficult and allows the defenses based on fuzzing the timers to be used. Going a step further, the hardware can also equalize the timing of each `rdseed` request by maintaining only a single 64-bit random seed inside the RNG unit. When a request is made, the RNG unit will first generate another seed and only then return the original seed to the requesting process. This also eliminates failures and simplifies the RNG hardware design, removing the need for multiple CB entries. The drawback of this approach is increased delay of the `rdseed` instructions. However, since these instructions are typically not used often, such an increase may be acceptable. Another potential problem is that it is not clear how the increased delays of some instructions impact interrupt handling and the application's response to interrupts.

5.2.2 *Partitioning RNG Resources*

If the performance impact of the equalization-based approach described above is too high, or significant issues with interrupt handling arise, an alternative mechanism can also be implemented in hardware, where vulnerable hardware resources of the RNG module can be fairly partitioned among the CPU cores. Specifically, each core can be assigned a slice of the CB and can only operate on that slice. As a result, the RNG activity performed by one process (the trojan) scheduled on one core will not impact a successful completion of an `rdseed` instruction issued by another process (the spy) executed on another core. Consequently, the RNG channel disappears. This approach requires a larger number of entries in the CB, but supports high performance and security. Partitioning the CB bits closes all inter-core channels, only leaving the possibility of a channel when the trojan and the spy are consecutively scheduled on the same core. To prevent such a channel, the RNG logic must ensure that the CB slice belonging to the core on which the communicating processes execute is completely refilled during the context switch interval. Fortunately, this is already the case with the current RNG implementation. According to our evaluation, the complete CB refill takes about 1.75 microseconds, while the context switch interval is typically around 3-4 microseconds [33, 47].

6. RELATED WORK

Several covert channels through shared processor resources have been described and analyzed in recent literature. In this section, we review some of these related works and contrast them with the RNG channel described in this paper.

The work of [56] presented a cross-VM covert channel through the L2 cache. According to the results, the L2 channel has a theoretical capacity of 262 bit/s, which is significantly lower than the capacity of the RNG channel. Under realistic loads in EC2 cloud, the mean bit rate was reported to drop further to 3.2 bit/s with the error rate of above 9%. Since the L2 cache is private to each core, the channel can only be used when two virtual machines share the same CPU. Core migration drastically reduces the channel's capacity.

Maurice et al. [39] developed a cross-core covert channel through shared Last Level Caches (LLC). The authors demonstrated the bit rate of about 1300 bit/s for a non-virtualized setup, and 751 bit/s for a virtualized setup. Setting up the channel through the LLC requires significant effort and understanding of complex LLC addressing nuances. In contrast, the RNG channel only requires simple manipulations with RNG instructions and trivial knowledge about the details of the RNG module operation.

The work of [55] presented a timing covert channel between virtual machines exploiting contention on the memory bus through the use of atomic memory instructions. The authors achieved the channel transmission rate over 700 bit/s in their laboratory setup and over 100 bit/s in Amazon EC2 cloud. This channel works in cross-core and cross-VM setting.

All channels created through caches and memory bus require access to either processor timekeeping infrastructure or to the hardware performance counters to allow the spy to time the events and recognize transmission of bits. However, the access to such infrastructure is not always available at the user level. In addition, such access can be disabled just to secure systems against such channels. Wang et al. [52] analyzed timing channels through shared memory controllers and proposed techniques to close this channel. Hunger et al. [22] present an excellent summary of multiple covert channels, including the ones through caches, memory bus, branch predictor and the AES hardware. They also analyze channel capacities and discuss synchronization protocols. [14, 12] perform a detailed analysis of covert channels through branch prediction units. The branch predictor channels only work if the trojan and the spy execute on the same physical core, either in two hardware thread contexts of a simultaneously multithreaded processor, or consecutively scheduled on a single-threaded core. Because the channel is on a single core, no additional synchronization is required in this case.

CC-Hunter [7] is a technique that detects the presence of covert timing channels by dynamically tracking conflict patterns on shared processor hardware. In principle, if CC-Hunter can be repurposed to track the `rdseed` events, it is conceivable that it will detect the RNG-based channel, but further investigations are needed to establish that.

Besides exploiting the shared physical CPU structures as covert communication medium, recent work also investigated the use of thermal characteristics of the CPU for secret data transmission [38, 2]. This example demonstrates that unexpected covert channels can be created and it is important to investigate, discover and mitigate all sources of such covert communications. This paper makes a contribution in this direction by uncovering a new and powerful covert channel and proposing software and hardware mitigations for it.

7. CONCLUDING REMARKS

In this paper, we introduced a new covert communication channel using the shared hardware random number generator unit as communication medium. The key idea is to control the pressure on the shared RNG hardware unit by the sender by either executing a sequence of `rdseed` instructions or busy-waiting, thus impacting the behavior of the `rdseed` instructions issued by the receiver. We showed that this channel works reliably and with high transmission rate across cores and across virtual machines and can be implemented directly within user space with no OS assistance. In addition, we showed that the RNG channel can be created without relying on any processor timekeeping infrastructure or hardware performance counters.

The above characteristics make the RNG channel easy to establish and difficult to mitigate. In terms of protection, we proposed two software approaches and two hardware approaches. The software approaches either utilize support available on Intel virtualization platforms to handle the `rdseed` instructions in special ways or executing the additional thread to create constant pressure on the hardware RNG unit. The hardware approaches are based on either equalizing the timing of the `rdseed` instructions across threads, or partitioning the RNG resources among cores to remove the dependency of instruction latencies issued by one thread on the instructions of the other.

For secure system design, it is important to develop architectures that do not have paths for information leakage, either through side channels or covert channels. To this end, it is critical to discover the new vulnerabilities in the existing and emerging systems and propose defenses against them. This paper makes contributions on both of these fronts.

8. ACKNOWLEDGMENT

We would like to thank our shepherd Edward Suh for his insight and suggestions for improving the paper, and the anonymous reviewers for their useful comments. This material is based on research sponsored by the National Science Foundation grant CNS-1422401.

9. REFERENCES

[1] AMD. AMD64 architecture programmer's manual volume 2: System programming, 2016.

[2] Bartolini, D. B., Miedl, P., and Thiele, L. On the capacity of thermal covert channels in multicores. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 24.

[3] Bello, L. DSA-1571-1 OpenSSL Predictable random number generator, 2008. Debian Security Advisory.

[4] Berger, S., Cáceres, R., Pendarakis, D., Sailer, R., Valdez, E., Perez, R., Schildhauer, W., and Srinivasan, D. TVDc: managing security in the trusted virtual datacenter. *ACM SIGOPS Operating Systems Review 42*, 1 (2008), 40–47.

[5] Bucci, M., Germani, L., Luzzi, R., Trifiletti, A., and Varanonuovo, M. A high-speed oscillator-based truly random number source for cryptographic applications on a smart card ic. *Computers, IEEE Transactions on 52*, 4 (2003), 403–409.

[6] Burdonov, I., Kosachev, A., and Iakovenko, P. Virtualization-based separation of privilege: working with sensitive data in untrusted environment. In *Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems* (2009), ACM, pp. 1–6.

[7] Chen, J., and Venkataramani, G. CC-hunter: Uncovering covert timing channels on shared processor hardware. In *Intl. Symp. on Microarchitecture* (2014), IEEE Computer Society, pp. 216–228.

[8] Demchik, V. Pseudo-random number generators for monte carlo simulations on ati graphics processing units. *Computer Physics Communications 182*, 3 (2011), 692–705.

[9] Dorrendorf, L., Gutterman, Z., and Pinkas, B. Cryptanalysis of the random number generator of the windows operating system. *ACM Transactions on Information and System Security (TISSEC) 13*, 1 (2009), 10.

[10] Evtyushkin, D., Elwell, J., Ozsoy, M., Ponomarev, D., Abu-Ghazaleh, N., and Riley, R. Iso-x: A flexible architecture for hardware-managed isolated execution. In *Proceedings of 47th International Symposium on Microarchitecture (MICRO)* (2014), pp. 190–202.

[11] Evtyushkin, D., Elwell, J., Ozsoy, M., Ponomarev, D., Ghazaleh, N. A., and Riley, R. Flexible hardware-managed isolated execution: Architecture, software support and applications. *IEEE Transactions on Dependable and Secure Computing (TDSC)* (2016).

[12] Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. Covert channels through branch predictors: a feasibility study. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (2015), ACM, p. 5.

[13] Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR. In *Proceedings of 49th International Symposium on Microarchitecture (MICRO)* (2016).

[14] Evtyushkin, D., Ponomarev, D., and Abu-Ghazaleh, N. Understanding and mitigating covert channels through branch predictors. *ACM Transactions on Architecture and Code Optimization (TACO) 13*, 1 (2016), 10.

[15] Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 627–638.

[16] Gruss, D., Maurice, C., Fogh, A., Lipp, M., and Mangard, S. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)* (2016), ACM.

[17] Gruss, D., Maurice, C., and Wagner, K. Flush+Flush: A stealthier last-level cache attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings* (2016), Springer.

[18] Guri, M., Monitz, M., Mirski, Y., and Elovici, Y. Bitwhisper: Covert signaling channel between air-gapped computers using thermal manipulations. In *Computer Security Foundations Symposium (CSF), 2015 IEEE 28th* (2015), IEEE, pp. 276–289.

[19] Gutmann, P. Software generation of practically strong random numbers. In *Usenix Security* (1998).

[20] Hamming, R. W. Error detecting and error correcting codes. *Bell System technical journal 29*, 2 (1950), 147–160.

[21] Hu, W.-M. Reducing timing channels with fuzzy time. *Journal of computer security 1*, 3-4 (1992), 233–254.

[22] Hunger, C., Kazdagli, M., Rawat, A., Dimakis, A., Vishwanath, S., and Tiwari, M. Understanding contention-based channels and using them for defense. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on* (2015), IEEE, pp. 639–650.

[23] Intel. Digital Random Number Digital Random Number Generator Generator (DRNG) Software Implementation Guide.

[24] Intel. Intel 64 and ia-32 software developer's manual, volume 3c: System programming guide, part 3.

[25] Jang, Y., Lee, S., and Taesoo, K. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)* (2016), ACM.

[26] Jun, B., and Kocher, P. The intel random number generator. *Cryptography Research Inc. white paper* (1999).

[27] Junkins, S. The Compute Architecture of Intel Processor Graphics Gen9.

[28] Kaplan, D., Kedmi, S., Hay, R., and Dayan, A. Attacking the Linux PRNG on android: weaknesses in seeding of entropic pools and low boot-time entropy. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)* (2014).

[29] Kayaalp, M., Abu-Ghazaleh, N., Ponomarev, D., and Jaleel, A. A high-resolution side-channel attack on last-level cache. In *Proceedings of the 53rd Annual Design Automation Conference* (2016), ACM, p. 72.

[30] Koetter, R., and Kschischang, F. R. Coding for errors and erasures in random network coding. *Information Theory, IEEE Transactions on 54*, 8 (2008), 3579–3591.

[31] Latham, D. C. Department of defense trusted computer system evaluation criteria. *Department of Defense* (1986).

[32] Lenstra, A., Hughes, J. P., Augier, M., Bos, J. W., Kleinjung, T., and Wachter, C. Ron was wrong, whit is right. Tech. rep., IACR, 2012.

[33] Li, C., Ding, C., and Shen, K. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science* (2007), ACM, p. 2.

[34] Li, K., Kavčić, A., Venkataramani, R., and Erden, M. F. Channels with both random errors and burst erasures: Capacities, ldpc code thresholds, and code performances. In *Information Theory Proceedings (ISIT), 2010 IEEE International Symposium on* (2010), IEEE, pp. 699–703.

[35] Liberty, J. S., Barrera, A., Boerstler, D. W., Chadwick, T. B., Cottier, S. R., Hofstee, H. P., Rosser, J. A., and Tsai, M. L. True hardware random number generation implemented in the 32-nm SOI POWER7+ processor. *IBM Journal of Research and Development 57*, 6 (2013), 4–1.

[36] Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy* (2015), pp. 605–622.

[37] Martin, R., Demme, J., and Sethumadhavan, S. Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. *ACM SIGARCH Computer Architecture News 40*, 3 (2012), 118–129.

[38] Masti, R. J., Rai, D., Ranganathan, A., Müller, C., Thiele, L., and Capkun, S. Thermal covert channels on multi-core platforms. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), pp. 865–880.

[39] Maurice, C., Neumann, C., Heen, O., and Francillon, A. C5: cross-cores cache covert channel. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, pp. 46–64.

[40] McKeen, F., Alexandrovich, I., Berenzon, A., C.Rozas, Shafi, H., Shanbhogue, V., and Svagaonkar, U. Innovative instructions and software model for isolated execution. In *Wkshp. on Hardware and Architectural Support for Security and Privacy, with ISCA'13* (2013).

[41] Moser, S. M. Error probability analysis of binary asymmetric channels. *Dept. El. & Comp. Eng., Nat. Chiao Tung Univ* (2009).

[42] Mundada, Y., Ramachandran, A., and Feamster, N. Silverline: Data and network isolation for cloud services. In *HotCloud* (2011).

[43] Naghibijouybari, H., and Abu-Ghazaleh, N. Covert Channels on GPGPUs. *Computer Architecture Letters* (2016).

[44] Pessl, P., Gruss, D., Maurice, C., Schwarz, M., and Mangard, S. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association, pp. 565–581.

[45] Sailer, R., Valdez, E., Jaeger, T., Perez, R., Van Doorn, L., Griffin, J. L., Berger, S., Sailer, R., Valdez, E., Jaeger, T., et al. sHype: Secure hypervisor approach to trusted virtualized systems. *Techn. Rep. RC23511* (2005).

[46] Schneider, F. B. Least privilege and more. In *Computer Systems*. Springer, 2004, pp. 253–258.

[47] Sigoure, B. How long does it take to make a context switch, 2010.

[48] Son, S. H., Mukkamala, R., and David, R. Integrating security and real-time requirements using covert channel capacity. *Knowledge and Data Engineering, IEEE Transactions on 12*, 6 (2000), 865–879.

[49] Stone, J. E., Gohara, D., and Shi, G. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering 12*, 1-3 (2010), 66–73.

[50] Sunar, B., Martin, W. J., and Stinson, D. R. A provably secure true random number generator with built-in tolerance to active attacks. *Computers, IEEE Transactions on 56*, 1 (2007), 109–119.

[51] Vattikonda, B. C., Das, S., and Shacham, H. Eliminating fine grained timers in xen. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop* (2011), ACM, pp. 41–46.

[52] Wang, Y., Ferraiuolo, A., and Suh, G. E. Timing channel protection for a shared memory controller. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on* (2014), IEEE, pp. 225–236.

[53] Wang, Z., and Lee, R. Covert and side channels due to processor architecture. In *Annual Computer Security Applications Conference* (2006), IEEE.

[54] Wicker, S. B., and Bhargava, V. K. *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[55] Wu, Z., Xu, Z., and Wang, H. Whispers in the hyper-space: high-speed covert channel attacks in the cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (2012), pp. 159–173.

[56] Xu, Y., Bailey, M., Jahanian, F., Joshi, K., Hiltunen, M., and Schlichting, R. An exploration of l2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop* (2011), ACM, pp. 29–40.

[57] Xu, Y., Cui, W., and Peinado, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems.

[58] Zhang, Y., Juels, A., Oprea, A., and Reiter, M. K. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Proc. 2011 IEEE Symposium on Security and Privacy (S&P)* (2011), pp. 313–328.