Constructing and Characterizing Covert Channels on GPGPUs

Hoda Naghibijouybari
University of California, Riverside
hnagh001@ucr.edu

Khaled N. Khasawneh
University of California, Riverside
kkhas001@ucr.edu

Nael Abu-Ghazaleh
University of California, Riverside
nael@cs.ucr.edu

ABSTRACT

General Purpose Graphics Processing Units (GPGPUs) are present in most modern computing platforms. They are also increasingly integrated as a computational resource on clusters, data centers, and cloud infrastructure, making them possible targets for attacks. We present a first study of covert channel attacks on GPGPUs. GPGPU attacks offer a number of attractive properties relative to CPU covert channels. These channels also have characteristics different from their counterparts on CPUs. To enable the attack, we first reverse engineer the hardware block scheduler as well as the warp to warp scheduler to characterize how co-location is established. We exploit this information to manipulate the scheduling algorithms to create co-residency between the trojan and the spy. We study contention on different resources including caches, functional units and memory, and construct operational covert channels on all these resources. We also investigate approaches to increase the bandwidth of the channel including: (1) using synchronization to reduce the communication cycle and increase robustness of the channel; (2) exploiting the available parallelism on the GPU to increase the bandwidth; and (3) exploiting the scheduling algorithms to create exclusive co-location to prevent interference from other possible applications. We demonstrate operational versions of all channels on three different Nvidia GPGPUs, obtaining error-free bandwidth of over 4 Mbps, making it the fastest known microarchitectural covert channel under realistic conditions.

CCS CONCEPTS

• **Security and privacy** → **Security in hardware**; **Hardware attacks and countermeasures**; **Hardware reverse engineering**; • **Computer systems organization** → **Single instruction, multiple data**;

KEYWORDS

GPUs, security, covert channels

ACM Reference format:
Hoda Naghibijouybari, Khaled N. Khasawneh, and Nael Abu-Ghazaleh. 2017. Constructing and Characterizing Covert Channels on GPGPUs. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 13 pages.
https://doi.org/10.1145/3123939.3124538

1 INTRODUCTION

General Purpose Graphical Processing Units (GPGPUs) are used to accelerate a range of applications including security, computer vision, computational finance, bio-informatics and many others [24]. Despite their improving performance and increasing range of applications, the security vulnerabilities of GPGPUs have not been well studied; only a few research papers have examined security vulnerabilities of GPGPUs [16, 19, 29, 30, 33, 46]. Often applications that use GPGPUs operate on sensitive data [4, 7, 25], which can be compromised by security vulnerabilities present in GPGPUs. Covert channel attacks are dangerous because they allow intentional communication of sensitive data between malicious processes that have no direct channel between them. A covert channel attack may enable a malicious application without network access to communicate data to another application to exfiltrate the data off the device. Alternatively, covert communication can be used to bypass protections that track exposure of sensitive information such as sandboxing or information flow tracking, allowing sensitive data to escape containment [10]. The presence of a covert channel can also forecast the possibility of a side-channel attack [32], although we do not pursue such attacks in this paper.

With multiprogramming starting to be available on GPUs [41, 44], covert channel attacks between two kernels running concurrently on a GPU become possible. The attack offers a number of advantages that may make them an attractive target for attackers compared to CPU covert channels including: (1) With GPU-accelerated computing available on major cloud platforms such as Google Cloud Platform, IBM cloud, and Amazon web service [26] this threat is substantial [35]. The model of sharing GPUs on the cloud is evolving but allowing sharing of remote GPUs is a possibility [3, 8, 31, 34]. Therefore, GPU covert channels may provide the attackers with additional opportunities to co-locate, which is a pre-requisite for these types of attacks [35]; (2) GPGPUs operate as an accelerator with separate resources that do not benefit from protections offered by an Operating system. In fact, due to this property they have been proposed for use as a secure processor [37]; and (3) GPGPU channels can be of high quality (low noise) and bandwidth due to the inherent parallelism and, as we demonstrate, the ability to control noise.

At the same time, constructing covert channels on GPGPUs introduces a number of challenges and operational characteristics different from those on CPUs. One of the new challenges is to how to establish co-location between the kernel and the spy by exploiting the hardware schedulers such that the communicating kernels can share resources. Thus, we first reverse engineer the hardware scheduling algorithms that determine where the different blocks and warps can be allocated to create contention (Section 3). A second problem is to identify which resources are most effective for communication given the throughput bound nature of GPGPUs. In particular, GPUs have substantial parallelism, which may enable high throughput covert communication, but only if effective isolated contention domains can be found. For example, a shared resource, such as the L2 cache, may have limited capacity limiting the bandwidth of communication through it. Alternatively, a resource may have high capacity (such as the memory bandwidth), making it difficult to create measurable contention. We construct different channels using contention on caches, contention for computational units, as well as contention for memory operations in Sections 4, 5 and 6 respectively. We discover that contention on the functional units is isolated by different warp schedulers: only the warps on the same warp scheduler experience contention. Thus, to communicate, the kernels must be mapped to the same warp scheduler, and conversely, kernels on different warp schedulers can construct parallel covert channels to increase bandwidth.

Having demonstrated and characterized these different channels on three different GPGPUs, we explore improvements to the channel bandwidth. We use the inherent parallelism in GPGPUs to increase the bandwidth of the channel. We also implement synchronization to increase the robustness of the communication to increase the communication efficiency in Section 7. To improve resilience to interference from other applications, we propose exploiting the hardware schedulers to block out co-location from other applications; this is a new approach to managing noise unique to GPGPUs (Section 8). We demonstrate the success of exclusive co-location on current GPGPUs which support multiprogramming based on leftover policy and discuss some scenarios to extend our attack for proposed multiprogramming schemes on future GPGPUs, such as kernel preemption and intra-SM partitioning. We discuss the potential mitigations in Section 9. Finally, Section 10 reviews related work, while Section 11 presents some concluding remarks.

The high level contribution of the paper is the first detailed exploration of covert channel attacks on GPGPUs, including the construction, characterization and optimization of different channels. In particular, the paper makes the following contributions.

- Reverse engineering co-location: We develop a methodology to reverse engineer the hardware scheduling algorithms and exploit them to force colocation to enable covert communication.
- Characterizing contention behavior: We study contention on functional units, memory resources, and different levels of the cache. We discover a number of subtle characteristics that significantly impact the operation of the channels.
- Demonstrating practical channels on three different GPGPUs. We discuss how the attacks generalize to recent GPU multiprogramming proposals.
- We propose techniques to force exclusive co-location of spy and trojan on current GPGPUs to prevent interference of other workloads, achieving noise-free communication.
- Optimizing Bandwidth of the channels: we exploit parallelism at different levels (SM, warp scheduler, cache sets) to increase bandwidth. We use synchronization to improve bandwidth and robustness. Together, we demonstrate on real hardware the highest known covert channel bandwidth.

2 BACKGROUND AND THREAT MODEL

Modern GPGPUs consist of a number of programmable streaming multiprocessors (SM, or SMX) that together access a shared global device memory. Figure 1 presents an architecture overview of a typical GPGPU. A program (kernel) may be broken into one or more blocks that are assigned to one or more of the SMs. Each block consists of a number of threads that are grouped into warps of typically 32 threads that are scheduled together using the Single Instruction Multiple Thread (SIMT) model. The warps are assigned to one of a typically few warp schedulers on the SM. In each cycle, each warp scheduler can issue one or more instructions to the available cores. Depending on the architecture, each SM has a fixed number of different types of cores such as single precision cores, double precision cores, and special functional units. Depending on the number of available cores an instruction takes one or more cycles to issue, but the cores are heavily pipelined making it possible to continue to issue new instructions to them in different cycles. Warps assigned to the same SM compete for access to the processing cores. In addition, warps assigned to the same warp scheduler may compete for the issue bandwidth of the scheduler.

Each SM has a large register file with a fixed number of registers assigned to each thread. The SM also has a shared L1 cache, and a shared memory region that is explicitly managed by the program. The device memory is connected to the chip using several high speed channels, resulting in bandwidths of several hundred gigabytes per second, but with a high latency. The impact of the latency is hidden partially using caches, but more importantly, the large number of warps/threads ensures the availability of ready warps to take up the available processing bandwidth when other warps are stalled waiting for memory. The memory is partitioned into global memory, constant memory and texture memory that are each shared among all SMs. They are also used to transfer data between the CPU and GPU. There is a shared L2 cache to provide faster access to memory. Each SM has its own on-chip shared memory, as well as several L1 caches for the instructions, global data, constant data and texture data. As a typical example, one of our target devices, the Nvidia Tesla K40C, includes 15 SMs, each featuring 192 single-precision CUDA cores. Each SM uses four warp schedulers and eight instruction dispatch units [42]. The size of the global memory, L2 cache, constant memory and shared memory are 12 GB, 1.5 MB, 64 KB and 48 KB respectively.

The current generation of GPUs supports multiprogramming, or the ability to run multiple programs at the same time, through multiple streams with multi-kernel execution within the same process, or a multi-process service (MPS) [27], which allows concurrent kernels from different processes. MPS is already supported on GPUs with hardware queues such as the Hyper-Q support available on Kepler and newer microarchitecture generations from Nvidia. Multiprogramming is required for our threat model since the spy and the trojan are different programs that are trying to communicate indirectly while running concurrently on the same GPU. To provide a uniform implementation including Fermi GPUs, we utilized streams for multiprogramming on GPU. Recent research has shown that there are significant performance advantages to supporting multiprogramming on GPUs [36, 41, 44]. For these reasons we believe that GPU manufacturers are moving to support multiprogramming; for example, it is supported by the Vulkan API which is intended to replace both graphics standards such as OpenGL and GPU compute standards such as CUDA [28].

Our threat model consists of a standard covert communication scenario with a trojan and spy kernels from two different applications

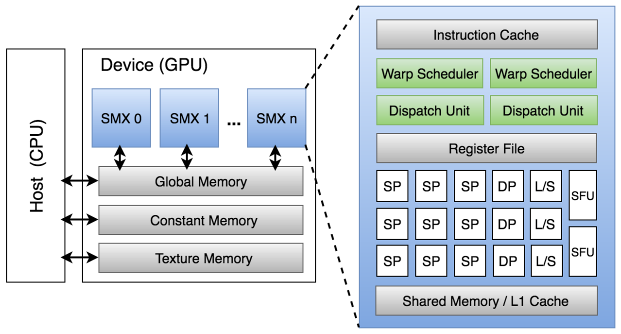


Figure 1: GPGPU architecture overview

that co-exist concurrently on the same GPU. The two kernels wish to communicate covertly. We consider a case where the trojan and the spy are the only two applications running on the GPU to characterize the bandwidth of the channels under the best case scenario. We later relax that assumption and explore approaches to prevent or tolerate noise from other applications. We assume that the two kernels can launch applications to the same GPGPU; in a cloud setting a first problem is to establish this ability. Since no standard sharing model of GPGPUs in the cloud has emerged, we do not focus on this problem [3, 8, 31, 34]. In most existing settings, the GPGPU is shared among applications on the same physical node, via I/O pass-through. In such settings, the problem boils down to achieving co-location on the same cloud node [35]. In the case of non-cloud scenarios, typically a GPGPU is also shared among applications on the same machine/device.

3 ESTABLISHING CO-LOCATION

Our goal is to create covert channels through contention on shared hardware resources in the GPGPU. As a first step, we need to control the placement of concurrent applications –in our case, the trojan and the spy. The placement defines what resources the applications share and therefore what covert channels are available for use. In this section, we show how to establish co-location as the first step in constructing covert channels. In particular, we reverse engineer the block assignment algorithm on real GPUs and show how to exploit it to establish co-location. It is likely that future GPGPUs may use alternative placement algorithms; however, we believe that the reverse engineering approach we use can be used for not only Nvidia GPUs, but also a large class of placement algorithms that are both deterministic and do not use preemption. We support this claim by considering the scheduling algorithms for other multiprogrammed GPUs that were recently proposed in literature [41, 44].

3.1 Co-location on existing GPGPUs

The Nvidia thread block assignment and kernel co-location algorithms are unpublished; thus, it is necessary to reverse engineer the placement algorithm. First, we explore whether blocks belonging to two kernels can be co-located on the same SM. We launch two kernels on different streams. In each kernel, we read the SM ID register (`smid`) for each block to determine the ID of the SM on which the kernel is running. In addition, we use the `clock()` function to measure the start time and stop time of each block. By using this information, and repeating the experiment for different numbers and configurations of blocks, we reverse engineered the placement algorithm.

We found that the blocks for the first kernel are assigned to different SMs in a mostly round-robin manner. If there are SMs that are idle, or that have a leftover capacity, they can be used for blocks of the second kernel, again in a mostly round-robin assignment. Otherwise, the blocks of the second kernel are queued until at least one SM is released. Therefore, if each kernel is launched with a number of blocks equal to or exceeding the number of SMs on the device, such that each block does not exhaust the resources of the SM, they achieve co-residency within an SM. This multiprogramming mechanism on current GPUs is called the Leftover policy.

We also discovered that there is another level of sharing within the SM that impacts the contention behavior. In particular, on many GPGPUs, there are a number of warp schedulers available on each SM. Each warp is associated with one of these warp schedulers. If different warps share the same scheduler, we show that their contention behavior is different since the warps on the same scheduler compete for the issue bandwidth that is assigned to the scheduler.

We experimented with the assignment algorithm of warps to warp schedulers and discovered that it is also round robin. With this knowledge, the spy and the trojan can set up their kernel parameters to achieve co-location on the same SM and if desired on the same warp scheduler. For example, on the Tesla K40C, with 15 SMs and 4 warp schedulers, if each of the spy and the trojan launch a kernel with 15 blocks each using 4 warps (i.e., 128 threads), they will each have a warp on each of the warp schedulers of each of the 15 SMs on the GPGPU.

3.2 Co-location on other GPGPUs

The left-over policy allows co-location of kernels opportunistically. Recent papers [41, 44] improve multiprogramming on GPUs using intra-SM resource partitioning that execute multiple kernels to more effectively utilize the GPU. We believe that this approach of varying the configuration of launched kernels and observing how they are scheduled can be used to reverse engineer any deterministic and non-preemptive co-location algorithm. In this section, we consider the co-location problem relative to these proposed schedulers.

Wang et al. [41] support a simultaneous multi-kernel by fine grained context switching at the granularity of thread block. To schedule thread blocks of the new kernel to an SM, those thread blocks of previously scheduled kernels that have the highest resource usage on the victim SM may be preempted. Thus, this scheduler makes co-location easier by allowing the spy and the trojan to reside on the same SM even if other applications are already running there. By using just one thread block for each spy and trojan on each SM, the spy and trojan will be guaranteed not to be preempted. However, the co-location of other workloads on the same SM possibly adds noise to the covert channel. We discuss this issue in Section 8.

Xu et al. [44] propose a dynamic intra-SM resource partitioning that does not use preemption. Intra-SM partitioning attempts to co-schedule kernels that are compatible in their resource usage to the same SM. Since this multiprogramming scheme does not use preemption, we can force exclusive co-location of the two kernels by manipulating their initial behavior so that the scheduler finds them compatible.

We also consider a case where the two kernels cannot be co-located on the same SM, for example, in cases where there is no leftover capacity on the SMs used by the first kernel. In this case, covert communication is still possible through contention on resources that are shared between all SMs such as global memory or the L2 cache. This inter-SM covert channels can also be applied to some proposed GPU multiprogramming which allows executing multiple kernels only on disjoint sets of SMs on GPUs (i.e., no intra-SM scheduling). For example, Adriaens et al. [1] use inter-SM resource partitioning. Similarly, Tanasic et al. [36] propose kernel allocation at the granularity of the whole SM to support multiprogramming. It is likely that these multiprogramming mechanisms are not as efficient as intra-SM resource partitioning, due to the limited number of SMs and large overhead of context switching on GPUs. Moreover, multiprogramming at the granularity of the full SM cannot address resource under-utilization occurring within an SM.

4 CACHE COVERT CHANNELS

In this section, we present the first of three classes of covert channels we investigate in the paper: covert channels through the caches. We illustrate the principles using constant caches, but the attack applies to other caches on the system. We selected constant memory because the size of both the L1 and L2 caches is small allowing us to create contention easily.

The attack proceeds in two steps. First, an offline step uses the microbenchmarking approach introduced by Wong et al. [43] to infer the constant memory and cache characteristics at each level of the hierarchy. The second step is the communication step where we use contention to create the covert channel.

4.1 Attack Step I: Offline Characterization of Constant Memory Hierarchy

The parameters of the constant memory hierarchy at each level of the cache hierarchy can be extracted from latency measurement of loading different size arrays from constant memory using a strided access pattern. The cache is first warmed by accessing the array, which is subsequently accessed again while timing the accesses [43]. The size of the array is increased and the access latency observed.

Figure 2 and Figure 3 show the latency measurements for the L1 cache and L2 cache respectively on the Kepler Tesla K40C GPU. Each point on the figure represents an experiment with the array size shown on the x-axis. While the latency remains constant, the array fits in cache. When the array spills out of the cache, the latency starts increasing. First, the spill causes misses only in one set. As we keep increasing the array size, spills in additional sets occur: the number of steps in the figure is equal to the number of cache sets. The cache line size corresponds to the width of each step. From the cache size, number of cache sets and cache line size, we can calculate the cache associativity. For example, for the Kepler (Tesla K40) and Maxwell (Quadro M4000) GPUs we find that the constant memory L1 cache is 2kB, 4-way set associative with 64 byte cache line, while the L2 cache is 32kB 8-way set associative with 256 byte cache line. In the Fermi (Tesla C2075) GPU, constant memory L1 cache is 4kB, 4-way set associative with 64 byte cache line and L2 cache has the same parameters as in Kepler and Maxwell.

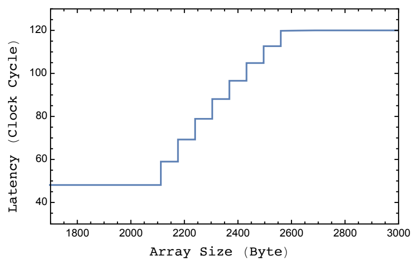


Figure 2: L1 constant cache, stride 64 bytes.

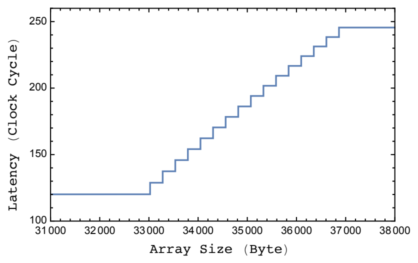


Figure 3: L2 constant cache, stride 256 bytes.

4.2 Covert channel through L1

We set up an experiment with two concurrent kernels using different streams on the GPU. On the Kepler K40C device, there are 15 SMs, so we use 15 thread blocks for each kernel to be sure of co-residency on the same SM. The trojan kernel communicates by either creating contention or doing nothing to encode 1 or 0 respectively. To create contention on one set, the trojan allocates an array with the size of L1 cache (2 KB) and loads it with a stride of 512 bytes to make the accesses hash into the same set. The spy also loads a 2KB array with the same stride as the trojan while timing the access: a high latency indicates 1 since the array was replaced by the trojan, and a low latency indicates a 0. Our results show that in the case of contention (i.e. sending 1), the measured latency by the spy is about 112 clock cycles, but without contention (i.e. sending 0), the latency is 49 clock cycles. This difference allows the spy to easily determine the bit being transmitted. Note that we create contention over only a single set of the cache, rather than over the whole cache, reducing the memory traffic and accelerating the attack.

To communicate multiple bits, the trojan and the spy have to stay synchronized. Due to scheduling variability and/or the presence of noise, loss of synchronization can occur. To simplify this problem in this experiment, we launch two kernels to communicate each bit of the message. Clearly, this incurs some overhead to launch the kernels, but it simplifies synchronization by leveraging the stream operations, resulting in error free bandwidth of around 40Kbps. In Section 7, we use synchronization through covert communication (on different sets of the cache) to remove the need to continue to relaunch the kernels. Implementing this synchronization makes the attack more robust to noise, and more resilient to loss of synchronization; it also improves the channel bandwidth significantly.

4.3 Covert channel through L2

When two kernels cannot be co-located on the same SM, they can still communicate through the L2 constant cache that is shared between all SMs. The process of creating a covert channel is the same as the L1 channel. However, the parameters of the L2 cache are different: we consider array size of 32kB and stride value of 4096 bytes (16 sets × 256 bytes) to fill just one cache set. The measured bandwidth in this scenario is about 20Kbps.

Figure 4 shows the bandwidth achieved by the attack on three Nvidia GPGPUs selected from three generations of microarchitecture (Fermi, Kepler and Maxwell). We modified the attack to fit the cache parameters, but otherwise left it unchanged. All bits were received correctly with no errors. Thus, high bandwidth covert channels are feasible through both levels of the cache.

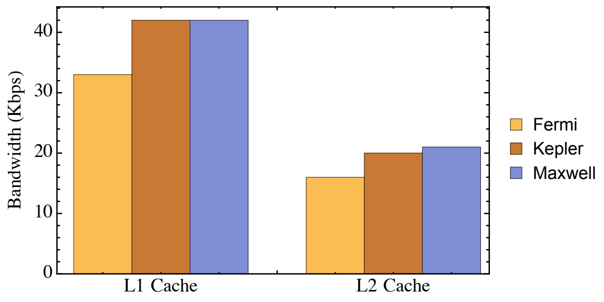


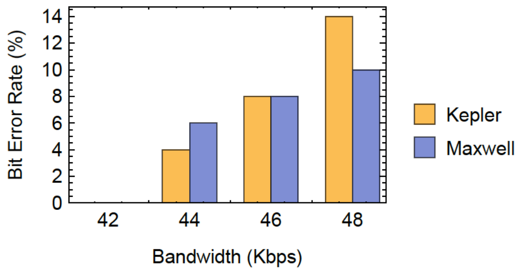
Figure 4: Cache channel bandwidth

Note that, to ensure overlap between spy and trojan processes and error free communication, we need to iterate sending each bit a sufficient number of times (20 times for L1 channel and 2 times for L2 Channel for Kepler GPU in our experiments). The minimum number of iterations is limited by two factors. First, without synchronization, we must ensure that sufficient iterations are present for the spy and trojan to overlap. Moreover, the clock() function on the GPGPU returns inconsistent results if the size of the code segment being timed is small. These factors place a limit on the minimum number of iterations necessary to detect contention.

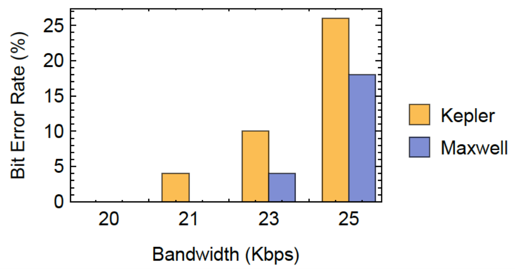
Decreasing the duration below 20 iterations causes errors, since the two kernels sometimes do not overlap. Figure 5 demonstrates the bit error rate as the bandwidth of channel increases (by decreasing the number of iterations) for the Kepler and Maxwell GPUs. The L1 and L2 channels on Fermi GPU also show nearly identical behavior around the reported error-free bandwidth.

5 FUNCTIONAL UNIT CHANNELS

Next we explore covert channels that use contention on the functional units (FUs) of the GPGPU. Conceptually, measurable contention can be created on functional units: when two kernels issue instructions to the same functional units, each should observe these instructions to execute slower than if either of them was issuing instructions



(a) L1 Channel



(b) L2 Channel

Figure 5: Bit error rate of L1 and L2 cache channels

on its own. While this general intuition holds true, we discover that contention behavior is significantly more complicated. The functional units are pipelined, isolating the contention to contention on the initial dispatch of operations to the functional units. This behavior is also moderated by the warp schedulers: we discover that mostly contention is isolated to warps belonging to the same warp scheduler, which must compete for the issue bandwidth of this scheduler. We first characterize the contention behavior, then show how we can exploit it to construct covert channels.

5.1 Step I: Characterizing contention behavior on the functional units

We set up an experiment to characterize the impact of contention on the performance of the different types of functional units. Each GPGPU has a number of computational cores that are specialized for different instructions; these include single precision units (SP), double precision units (DPU), and special function units (SFU). The number of functional units varies by the architecture and the type of the functional unit. We show only floating point operations because they have the most stringent issue limitations making it easier to create contention, and achieving the highest bandwidth.

For different types of operations, we launch just one kernel which executes a fixed number of operations to the functional unit being characterized. We increase the number of warps and measure the latency of each operation. Figure 6 presents latency plots for different single precision floating point operations (`__sinf` and `sqrt` which are executed on SFUs and Add and Mul which are executed on SPs) on different architectures. Figure 7 presents latency plots for double

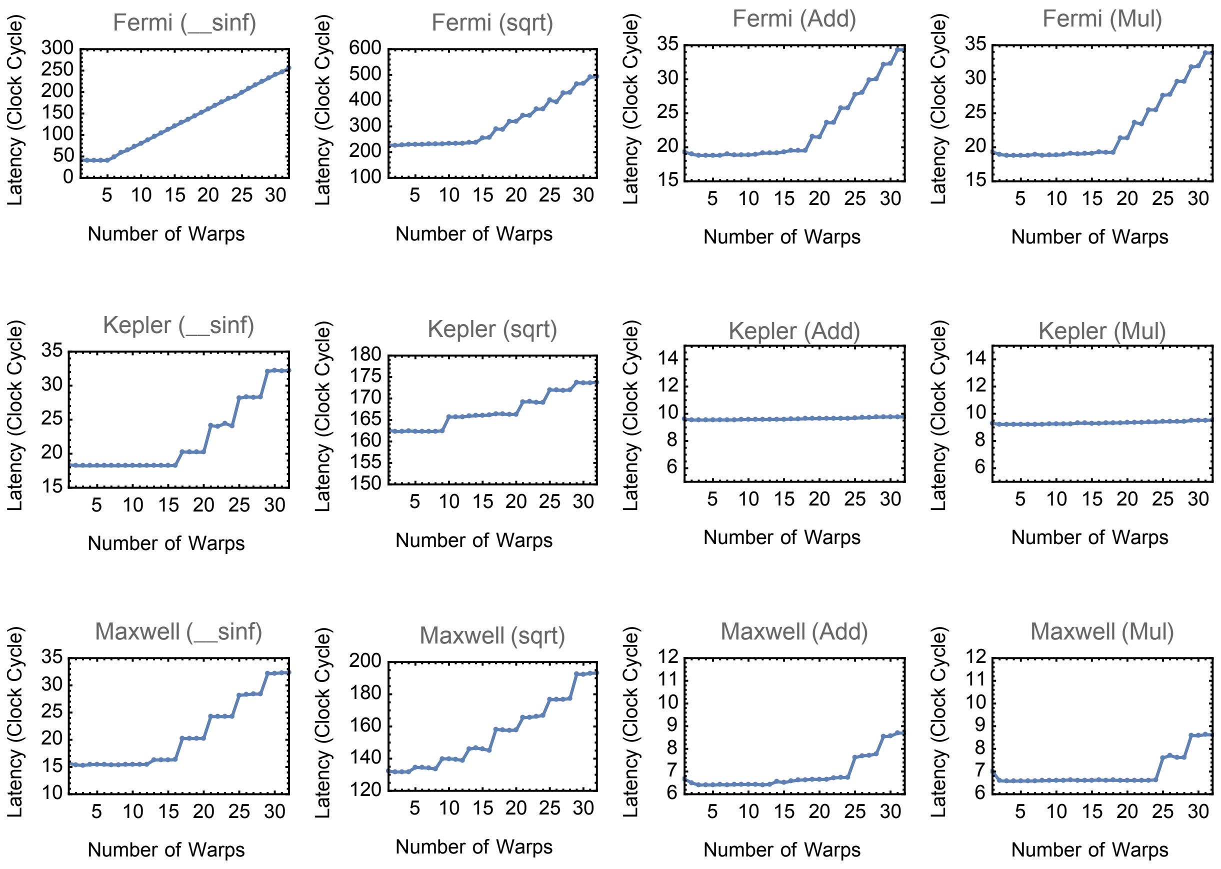


Figure 6: Latency of one single precision operation for different number of warps averaged over 128 iterations

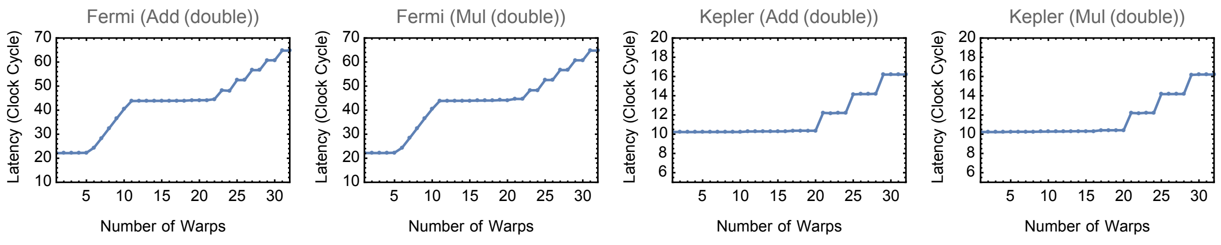


Figure 7: Latency of one double precision operation for different number of warps averaged over 128 iterations

precision Add and Mul on Fermi and Kepler GPUs (Maxwell GPU does not have double precision units).

The latency plotted in Figure 6 and Figure 7 is an approximate measure in this case since it is a function of not only contention but also the number of iterations of the experiment and the depth of the functional unit pipeline. The latency places an upper bound on the bandwidth of the channel since even if contention is possible with a single operation, the latency is the minimum delay of a communication cycle. However, the shape of the delay curve is more important since it establishes the degree of contention at which observable changes in measurable delay occur.

The number of available resources in each SM is shown in Table 1 for the three GPUs. In the Maxwell GPU (Figure 9), each SM is divided into quadrants, and each quadrant has its own registers, instruction buffer, and scheduler that spans 32 single-precision CUDA

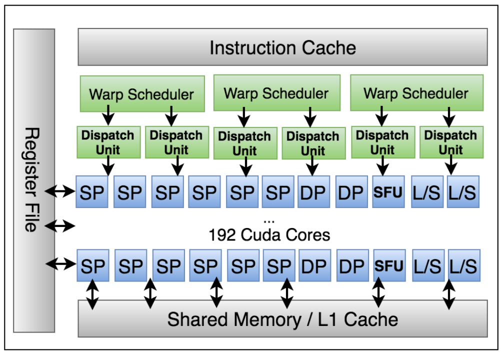


Figure 8: Kepler SM architecture overview

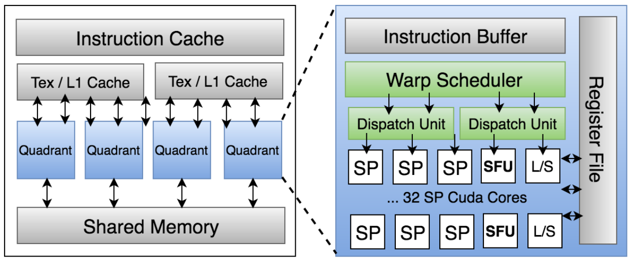


Figure 9: Maxwell SM architecture overview

cores and 8 SFUs. Each warp scheduler manages the resources for one of the quadrants. In contrast, the Fermi and Kepler GPUs (Figure 8) implement soft sharing where the warp schedulers do not have dedicated resources and instead issue instructions to a shared set of resources on the core.

Main Observations: Due to the different number of warp schedulers, the number of functional units of each type and the depth of the pipeline for each functional unit, we can see different behavior in each plot. The figures show the latency observed by warp 0 which is assigned to the first scheduler on the GPGPU. As we add warps, we see a step in latency plots, at the points where the number of operations causes contention either for scheduler issue bandwidth or for available functional units. For all experiments, the latency stays fixed as we increase the number of warps up to the point where the number of issued instructions matches the number of functional units for that operation. After that point, increasing number of warps leads to an increase in latency for the warps that are co-located on the same warp scheduler with the last added warp. *Contention is isolated to warps belonging to the same warp scheduler.* This result held even for the Kepler GPU which has soft sharing of the resources. For Kepler, the latency steps are not visible for single precision Add and Mul operations, due to the large number of available SP units. In contrast, in Maxwell the latency steps are eventually observable because the resources are split into quadrants (Figure 9).

5.2 Step II: Constructing FU covert channels

The trojan generates operations to the target functional unit to create contention when it desires to communicate 1, while it stays idle when it desires to communicate 0. Because of limited number of SFUs and clear jumps in latency on all three architectures, we elect to use contention on the SFUs. In particular, we take advantage of the more clear steps and lower latency of `__sinf` operation to create and demonstrate a covert channel through the warp schedulers and special functional units. Similar channels can be constructed using other resources.

We launch two concurrent kernels on different streams on GPU. To be sure that thread blocks of the two kernels can be co-located on each SM, we consider a number of blocks for each kernel equal to the number of SMs for different architectures. To be compatible with latency plots in Figure 6, we use the minimum number of required warps that will cause observable latency difference using the `__sinf` operation on the Spy side. This requirement translates to having each block of the spy and the trojan use 3 warps, 12 warps and 10 warps, for the Fermi, Kepler and Maxwell architectures respectively. The spy kernel does a number of `__sinf` operations which are executed on the SFUs. To send 0, the trojan does nothing so just 3 warps of the spy (or 12, 10, on the Kepler and Maxwell respectively) are scheduled to issue instructions and execute on SFUs. The latency in this case is about 41 clock cycles for Fermi (18 for Kepler and 15 for Maxwell) which is equal to the latency when there is no contention. For sending 1, the trojan executes a number of `__sinf` operations, so that its warps are scheduled to issue instructions alongside of the spy warps. In this case, there is contention on the SFUs and latency is increased to 48 clock cycles for Fermi (24 for Kepler and 20 for Maxwell). The measured covert channel bandwidth is 21 Kbps, 24 Kbps and 28 Kbps on Fermi, Kepler and Maxwell GPUs respectively.

6 GLOBAL MEMORY CHANNELS

In this section, we explore constructing covert channels through global memory, which provides an additional resource for contention when kernels are not co-located on the same SM. We also explore the impact of the access pattern on interference (coalesced vs. uncoalesced addresses). In particular, Jiang et al. [14] demonstrated a side channel attack on GPUs that times an AES encryption kernel running on the GPU from the CPU side. This attack relies on an observation that key-dependent differences in the coalescing behavior of memory accesses, lead to key-dependent encryption times which are used to infer the secret key. Thus, we wanted to explore whether the same phenomena can be exploited to produce high quality covert channels inside the GPU.

Using normal load and store operations, we did not observe reliable contention in the global memory. We believe that this is due to the high memory bandwidth. In particular, to saturate the memory bandwidth, many global memory operations are required, each with high latency, significantly harming achievable bandwidth. To create contention, we focused on atomic operations. Since the atomic operations rely on atomic units that are limited in number, it is possible to cause measurable contention. At the same time, atomic operations are extremely slow, which can limit the bandwidth of

Table 1: Number of available resources in each SM.

GPU	Warp Scheduler	Dispatch Unit	SP	DPU	SFU	LD/ST
Tesla C2075 (Fermi)	2	2	32	16	4	16
Tesla K40C (Kepler)	4	8	192	64	32	32
Quadro M4000 (Maxwell)	4	8	128	0	32	32

the covert communication; nevertheless, this channel achieves comparable bandwidth to other inter-SM channels. We defined three scenarios to understand the observable contention in global memory as follows. As before, the spy executes the operation described in each scenario while the trojan executes the same operations to transmit 1, or does nothing to transmit 0. In all three scenarios, the spy and trojan kernels access two different arrays located in global memory.

- Scenario 1: Each thread does atomic additions on one particular global memory address. This address differs for different threads.
- Scenario 2: Each thread does atomic additions on strided global memory addresses. These addresses differ for different threads and accesses for all threads in each warp are **coalesced**.
- Scenario 3: Each thread does atomic additions on consecutive global memory addresses. These addresses differ for different threads and accesses for all threads in each warp are **uncoalesced**.

The bandwidth of the three scenarios for each of the three GPUs is shown in Figure 10. For each GPU, we tune the number of iterations to the minimum that will cause observable contention. On the Kepler and Maxwell GPUs the throughput of global memory atomic operation is significantly higher than that of the Fermi since atomic operations are supported through the L2 cache and due to the addition of more atomic units in hardware [42]. Atomic operation throughput to a common global memory access is improved by 9x to one operation per clock cycle causing the overall time to communicate a bit to decreased significantly. It is interesting to note that the coalescing behavior that benefited Jiang et al.'s attack can not be exploited to create the covert channel attack. Coalescing causes timing variability to a single kernel, benefiting an external adversary that times this kernel and has no effect on timing of a competing kernel. However, our experiments show that the coalescing behavior on GPUs improves the channel bandwidth. The poor coalescing significantly reduces the possibility of using the faster L2-level atomic operation support, significantly slowing down the covert communication across the two different kernels. We can see that scenario 3 results in the lowest achievable covert channel bandwidth.

7 IMPROVING THE ATTACK

In this section, we explore several improvements and optimizations to increase the bandwidth and reliability of the covert channels.

We use two general approaches to increase the bandwidth of a covert channel on a single resource: (1) identifying opportunities for parallelizing the communication so that multiple trojans are communicating to multiple spies concurrently; and (2) implementing

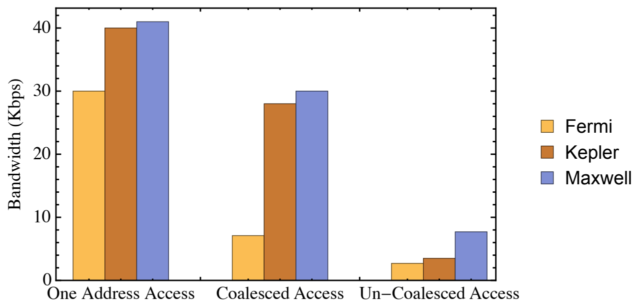


Figure 10: Global atomic covert channel bandwidth

synchronization to eliminate the loss of bandwidth that results from timing drift or, alternatively, the overhead of successively launching kernels. It is also possible to use multiple resources simultaneously to increase bandwidth. As an example, we experimented with sending two bits concurrently, one through L1 constant cache and one through the SFUs, achieving 56 Kbps bandwidth for Kepler and Maxwell GPUs. This approach is orthogonal to single-resource channel optimizations, allowing us to increase the bandwidth in those cases as well. We do not report multi-resource bandwidth since it is possible to use multiple resources on the CPU as well.

7.1 Improving the Cache Channels

The implementations we discussed so far are susceptible to loss of synchronization where the trojan and spy are not in sync with each other. This can occur due to natural drift due to unpredictable pipeline dependencies, or due to interference from other workloads. To overcome this issue, the covert channels we presented so far, forces overlap between the trojan and the spy by timing the launch of the kernel, leading to significant overhead and loss of bandwidth. Moreover, in the presence of competing applications, co-location may be difficult to achieve repeatedly.

To improve both the robustness and the bandwidth of the attack, in this section, we implement synchronization between the spy and the trojan through the covert channel. With synchronization, the two kernels are launched only once and use synchronization to communicate continuously. We illustrate the synchronization process for the L1 and L2 covert channels, but it is possible to implement synchronization for other channels as well.

The synchronized implementation uses three different sets of cache to fully synchronize sending and receiving bits, as follows. As with the basic cache side channel we use one set for communication. The two other sets are used to signal ready-to-send from the trojan to the spy and ready-to-receive from the spy to the trojan respectively.

$D_{Send}N, D_{Recv}N$: N is number of bits to transmit and receive
ReadyToSend(): sends ReadyToSendSig to Spy; ReadyToReceive(): sends ReadyToReceiveSig to Trojan
wait(S): while loop that breaks on signal S
prime(): fills the communication cache line; probe(): access the communication cache line and return 0 on a hit and 1 on a miss

Trojan protocol:	**Spy protocol:**	Description:
for $i \leftarrow 0$ **to** $N-1$ **do**	**for** $i \leftarrow 0$ **to** $N-1$ **do**	
ReadyToSend()	wait(ReadyToSendSig)	Handshake stage
wait(ReadyToReceiveSig)	ReadyToReceive()	
if $D_{Send}i = 1$ **then** prime()	$D_{Recv}i \leftarrow$ probe()	Transmitting and receiving stage

Figure 11: Synchronization communication protocol

The synchronized protocol is shown in Figure 11 and explained below.

On the Trojan side, for sending each bit:

- First, the trojan sends a ready-to-send signal by filling the pre-agreed on cache set with its data.
- Next it waits on the ready-to-receive cache set to ensure the spy is ready to receive the bit.
- Finally, we send the bit by either filling the set or not. The algorithm moves back to step 1.

Correspondingly, on the Spy side, for receiving each bit:

- First, the spy repeatedly checks the ready-to-send set to check if the trojan has sent the signal.
- Once it detects the ready-to-send signal (by measuring cache misses on the set), it sends the ready-to-receive signal on the corresponding set.
- Finally, we receive the bit through checking the access time observed on the communication set.

Infrequently, due to noise or other factors loss of synchronization can occur leading to deadlock where the spy and trojan are each waiting in a different part of the communication loop. To address this situation, we changed the algorithm to time out (by bounding the number of wait iterations) when the expected signal is not received in time. In the case of a timeout at the sender or the receiver, we regain synchronization by repeating the step prior to the wait. With this modification, the communication works seamlessly. We use a three way handshake to ensure that the trojan and spy are concurrently active at the bit communication component of the program; attempting a two way handshake led to noise and frequent loss of synchronization.

Although communication through three sets (rather than just one in the original channel) is required to fully synchronize the communication, removing the overhead of launching kernels for each bit of message increases the bandwidth to 61, 75 and 75 Kbps for the Fermi, Kepler and Maxwell GPUs, respectively.

To further improve the synchronized channel bandwidth, we utilize SIMT execution model of GPUs to send M bits through M different cache sets concurrently in each round. Two cache sets are used for signaling and we use the remaining cache sets for communication, such that one bit can be communicated through each cache set by different threads in parallel. For example, in Kepler and Maxwell GPUs L1 constant cache has 8 sets, enabling transfer of 6 bits concurrently. This parallelism increases the bandwidth to 207, 285 and 285 Kbps for the Fermi, Kepler and Maxwell GPUs, respectively. Our experiments demonstrate that by sending 2 bits, 4 bits and 6 bits concurrently, we are able to achieve 1.8x, 2.9x and 3.8x bandwidth improvement in Kepler GPU. Note that the ratio of bandwidth improvement is sublinear in the number of bits; we believe that this is due to both port contention, as well as the higher possibility of a cache miss in each kernel iteration.

The next approach to improve the bandwidth is to exploit the inter-SM parallelism available on the GPGPU. In particular, if we manage to colocate the trojan and the spy on multiple SMs, each of these instances can communicate independently using resources on the SM. With 15 SMs available in Tesla K40C device, the trojan is able to send 15 bits simultaneously and the bandwidth of communication is increased 15 times. Table 2 shows the covert channel communication bandwidth for the baseline attack (column 1), the attack with synchronization (column 2), the attack with synchronization and sending multiple bits through different cache sets (column 3) and the attack with all three improvements (column 4). Clearly, high quality, high bandwidth covert channels are feasible on GPGPUs.

For the L2 constant cache covert channel, we can create contention in parallel, either by filling the cache in parallel or by using each thread block to communicate through one particular L2 cache set. In theory, this should enable the trojan to send 16 bits (number of L2 cache sets) simultaneously. However, we observe only an 8x improvement in the best case, which we conjecture is due to cache port contention and cache bank collisions.

7.2 Improving the SFU Channel

Like the L1 covert channel, there is an opportunity to improve the bandwidth by having a spy and a trojan communicate on each SM for a 15x increase in bandwidth. However, the warp scheduler offers additional opportunities to increase the bandwidth. In Section 5, we also reverse engineered the warp assignment algorithm and found it to be round robin. By measuring latency for each warp, we noticed that by increasing number of warps one by one we see latency increasing *only in the warps that are assigned to the same warp scheduler*. This effect is likely due to the limit on the number of

Table 2: Improved L1 Channels

GPU	L1 Baseline	Sync.	Sync. and multi-bits	Sync., multi-bits and parallel
Fermi	33 Kbps	61 Kbps	207 Kbps	2.8 Mbps
Kepler	42 Kbps	75 Kbps	285 Kbps	4.25 Mbps
Maxwell	42 Kbps	75 Kbps	285 Kbps	3.7 Mbps

Table 3: Improved SFU covert channel communication bandwidth on different GPU architectures.

GPU	Baseline	Parallel through warp schedulers	Parallel through warp schedulers and SMs
Tesla C2075 (Fermi)	21 Kbps	28 Kbps	380 Kbps
Tesla K40C (Kepler)	24 Kbps	84 Kbps	1.2 Mbps
Quadro M4000 (Maxwell)	28 Kbps	100 Kbps	1.3 Mbps

instructions that can be dispatched by each warp scheduler. Certainly, this behavior is more clear in Maxwell GPUs in which each warp scheduler has its own functional units, but it is also present in the Kepler and the Fermi architectures where the functional units are soft-shared among the schedulers.

We use the observation that contention is isolated among the different warp schedulers to parallelize the covert channel attack by sending one bit through each warp scheduler concurrently. We consider K warps for spy which K is multiple of N (the number of warp schedulers) such that the latency of K warps alone is in the area of constant delay in the *__sinf* plots of Figure 6 for each architecture. For the trojan, we consider M warps such that M is also multiple of N and the latency of M+K is on one of the steps in the plot. Each warp of the trojan and the spy is assigned to one of the N different warp schedulers. We select one warp from each scheduler from the trojan to send one bit and another corresponding warp on the spy side to receive the bit. We are successfully able to parallelize the communication in this way, leading to a bandwidth that is N times more than the baseline channel. We are also able to carry out this communication independently on each of the S SMs, leading to increasing the bandwidth by another factor of S. The Covert channel bandwidth for the baseline channel is compared to the two parallelization steps (across SMs, and across warp schedulers in each SM) in Table 3. Note that the Fermi GPU has two warp schedulers per SM while Kepler and Maxwell have four warp schedulers per SM.

8 MITIGATING NOISE

We consider the presence of interference from other workloads which can affect the covert channel in two ways: (1) Co-location: the other workloads may prevent the spy and trojan from being launched together, or may cause them to be assigned to different SMs preventing the high quality covert channels that are present inside an SM; and (2) Noise: even if the spy and trojan are co-located, a third application may use the resources used for covert communication adding noise or even completely disrupting communication.

The high parallelism available on GPGPUs can result in high degrees of noise that challenge error detection and correction. Thus, our primary approach to manage noise is to try to prevent it completely by manipulating the block scheduler to achieve exclusive co-location between the spy and trojan at the level of the SM or the full GPGPU. We take advantage of concurrency limitations on current GPUs which use leftover policy for multiprogramming. In particular, we have the trojan and spy ask for resources in a way that they can be co-located with each other but that also makes it difficult for other applications to co-locate with them. For example, the spy may ask for the maximum available amount of the shared memory (or registers, thread blocks, threads, etc..) while the trojan asks for only the leftover amount of that resource. If there are not enough registers or shared memory available per multiprocessor to process at least one block, a competing kernel will be blocked until these resources become available. In this way, once the spy and trojan are launched, they can execute on the GPU (or SM) exclusively and communicate without interference.

In our Fermi (Tesla C2075) and Kepler (Tesla K40C) GPUs, the maximum shared memory per thread block is equal to maximum shared memory per SM (48KB). Since each of the spy and trojan needs just one thread block for communication, if the spy block asks for the maximum shared memory, it can saturate the SM in term of shared memory. In this case, the trojan block can be co-located on the same SM, if it does not use any shared memory. Any other application that uses shared memory, cannot be executed until the spy finishes and there are enough resources for thread blocks of the third application to be scheduled.

To improve the chances for exclusive co-location, we can also launch additional kernels that do not generate noise to exhaust other resources but do not use the resources that are claimed by the trojan and spy. If such kernels are launched at the same time as the kernel and spy, the scheduler will prefer to run them with the trojan and spy since it prioritizes kernels based on their launch time.

Note that, on our Maxwell GPU architecture (Quadro M4000), the maximum shared memory per SM is twice the maximum shared memory per thread block. So if the thread blocks of both trojan and spy ask for the whole shared memory we can exclusively run on each SM and noise is prevented.

To evaluate our exclusive co-location strategies, we executed the Rodinia benchmark applications [5] on a third stream, alongside the spy and trojan communication using L1 cache channel. By forcing exclusive co-location of the spy and trojan through saturating shared memory on each SM, we were able to prevent interference against all interfering workloads and workload mixtures and achieved error free communication in all cases. These workloads include applications which use shared memory and those that do not. They also include workloads such as *Heart Wall* that uses constant memory and that would interfere with the L1 covert channel if it were co-located with the malicious kernels. We note that if the interfering workload is launched before the spy and trojan, the built in synchronization in the kernels allows one of them to wait for the other. When the second kernel is launched, the resource request pattern ensures exclusive co-location after that.

As we discussed in Section 3, Wang et al.'s [41] intra-SM resource partitioning mechanism simplifies co-location of two malicious kernels on the same SM. However, it also allows other workloads to execute on the same SM, necessitating approaches for noise avoidance or tolerance. In contrast, the approach by Xu et al. [44] does not support preemption and thus allows exclusive co-location similar to current GPUs. When exclusive co-location to prevent noise is not possible, noise could be avoided or tolerated using techniques such as the following.

- Dynamically identifying idle resources: The approach is similar to whitespace communication in wireless networks where the radios opportunistically discover and use available channels without prior agreement [2], and solutions from that space can be leveraged for our problem. For example, the sender may scan through available resources (e.g. cache sets) in a pre-agreed on order until it discovers idle ones and transmits a beacon pattern on them. The receiver follows by scanning sets until it observes the beacon.
- Error correction: transmit error correcting codes with the data (sacrificing some of the bandwidth).

Since we were able to establish exclusive co-location on our GPUs and achieved noise free communication, we did not pursue either of those directions.

9 POSSIBLE MITIGATIONS

In this section, we provide a brief discussion of possible mitigations which we hope to explore in future work. One approach is to use partitioning to ensure possibly communicating applications do not have a way to effect measurable contention to each other. Partitioning can be done spatially (e.g., partitioning the cache [9, 17, 39], or temporally (e.g., ensuring instructions from different kernels do not execute in the same time period).

Specifically on GPUs, partitioning can be achieved at intra-SM and inter-SM level resources through scheduling of different application thread blocks or warps to separate them temporally or spatially. In addition, it is possible to fairly partition shared hardware resources among multiple simultaneous kernels based on their workload and resource requirements and make these partitions private to each application to eliminate interference. Although these approaches may lead to performance degradation and add some hardware overhead, they can prevent covert and side channel attacks which are results of unrestricted access to shared hardware resource from two or more co-located applications.

Another mitigation approach is to attempt to detect anomalous contention [6]. Given the different nature of GPGPU workloads and the degree of contention that is likely to arise naturally, a detailed evaluation of this class of solutions is necessary to assess its effectiveness. Solutions are possible that add entropy either to the assignment of the resources [40] or to the measurement of time [20]. Finally, scheduling algorithms that interfere with co-location, accommodate preemption, or prevent exclusive co-location (to introduce noise), can significantly complicate the attack.

10 RELATED WORK

Olson et al [30] developed a taxonomy of vulnerabilities and security threats for accelerators based on threat types (Confidentiality, Integrity, and Availability) and risk categories (what part of the accelerator they affect). Although the paper offers no concrete attacks, it highlights that security of accelerators warrants significant attention. Olson et al. [29] also propose sandboxing accelerators when the CPU and the accelerators share the same address space (e.g., under a Heterogeneous System Architecture configuration). This type of defense does not protect against side- and covert-channel vulnerabilities.

Lee et al. [16] reveal three major security threats in GPUs including: lack of initialization of newly allocated memory pages, un-erasable portion of GPU memory (e.g., constant data) and lack of prevention of threads of a kernel to access the contents stored in the local and private memories, written by threads of other kernels. When multiple users share the same GPU, there is information leakage between concurrently running processes or from processes that recently terminated. They utilize information leakage that occurs due to not clearing newly-allocated memory in the GPU to extract rearranged webpage textures of Chromium and Firefox web browsers (both use GPU-accelerated rendering) from Nvidia and AMD GPUs.

Similar attacks are presented by Di Pietro et al. [33] who also exploit non-zeroed state available in shared-memory, global-memory and registers. Maurice et al. [22] show that the attacks translate to virtualized and cloud computing environments where the adversary launches a virtual machine after the victim's virtual machine using the same GPU. This class of vulnerability can be closed by clearing memory before it gets reallocated to a different application.

Because GPGPUs have separate physical resources, PixelVault proposes using them as secure co-processors [37]. Recent work has shown that GPGPUs are vulnerable to disclosure attacks through the driver from a privileged user on the CPU, bringing into question the security of the PixelVault model [46]. Such attacks require root access and are outside our threat model.

Covert channels have been explored by many studies in the context of CPUs (e.g., [11, 13, 18, 21, 23, 32, 38]). For an overview, an excellent recent study characterized the bandwidth of several CPU contention-based covert channels including cache based covert channels [13] in a noisy setting. They derived a theoretical upper bound on capacity of practical channels and found the L1 covert channels (216 Kbps) and memory bus (565 Kbps) to be the two of highest upper bound capacity channels in theory. However, these are theoretical bounds and no channels under realistic conditions were demonstrated. Gruss et al. [12] study covert channels under the assumption that the attacker and the victim have accessed to shared memory pages (e.g., from shared libraries). They show a Flush+Reload attack with 2.3 Mbps and 0% error rate and a Flush+Flush attack with 3.9 Mbps and 0.84% error rate. Note that these attacks do not rely on contention and are not possible if there are no shared pages between the trojan and spy. These channels were also not tested under noisy environments, and therefore these results should be considered an upper bound on the achievable bandwidth. Maurice et al. [23] characterize noise on cache covert channels and propose an error handling protocol to build a reliable covert channel through which they can build an ssh connection. They obtained a bandwidth of 360 Kbps with 0% error rate for covert channel on Amazon EC2. The GPGPU covert channels we demonstrated obtain robust and error free communication with much higher bandwidth, by exploiting the inherent parallelism in GPGPUs, and controlling noise through exclusive co-location.

A few defenses against covert channels have appeared in the context of CPU channels. Chen et al. [6] proposed a framework to detect and mitigate timing covert channel on shared hardware resources on a CPU by monitoring conflicts. Hunger et al. propose a different approach to the detection of contention [13]. Yan et al. propose a record and replay framework that reconfigures shared resources during replay to detect unnatural contention indicative of covert channels [45].

Side-channel vulnerabilities are a dual of covert channel vulnerabilities where the leakage through the side channel is exploited by a spy to derive sensitive information from an executing victim; many CPU side channel attacks have been explored, including attacks on caches (e.g., [18, 32]). Several defenses and mitigations for these attacks have been proposed (e.g., [9, 17, 39, 40]). Two recent papers [14, 19] demonstrate the feasibility of side channel attacks on GPGPUs. Luo et al. [19] utilize a power-based side-channel attack on a GPU to gain secret information from an AES implementation being accelerated by the GPU.

Jiang et al. [14] conduct the first timing attack at the architecture level on GPU. Due to inherent SIMT and memory coalescing behavior present on GPUs, they identify the correlation between execution time of a single kernel and unique cache line requests. By calculating the number of unique cache line requests for different guessed keys and observing the correlation with the measured execution time, the correct key at the last round of AES encryption is extracted. The same group [15] presented another attack on table-based AES encryption. They found correlation between execution time of one table lookup of a warp and a number of bank conflicts generated by threads. They use these key-dependent differences in timing to correlate measured execution time to the key at the last round of the AES encryption as it executes on the GPU. Unlike the channels we explore in this paper, these timing channels are measured from the CPU side, limiting their bandwidth, and requiring each kernel to be able to launch (or anticipate the launch) of the other to time it.

We also found that the self-contention exploited [14] cannot be used for covert communication. Although memory coalescing and shared memory bank conflicts make a large difference in the timing of one kernel, these artifacts had little measurable effect on the timing of a competing kernel.

11 CONCLUDING REMARKS

This paper demonstrates the feasibility of covert channel communication on GPGPUs. We explored how the communicating kernels can leverage the block scheduler and then warp to warp scheduler mapping to achieve co-residency. Based on their co-residency options (on the same SM or across SMs), the SM local resources or inter-SM shared resources can be used for contention. We demonstrated attacks on constant caches, functional units and global memory on three generations of Nvidia GPUs. We explored different optimizations to increase the bandwidth using synchronization and available parallelism on the GPGPU. We also demonstrated preventing interference from other workloads (Rodinia benchmark) on covert communication. Our experiments on constructing covert channels through different shared hardware resources show that high bandwidth (over 4 Mbps) and error free covert communication is feasible on GPGPUs. Our future works will include exploring the possibility of side channel attacks on GPGPUs, as well as implementation of some mitigation techniques against covert- and side-channels.

12 ACKNOWLEDGEMENT

This material is based on research sponsored by the National Science Foundation grants CNS-1422401 and CNS-1619450.

REFERENCES

[1] Jacob T. Adriaens, Katherine Compton, Nam Sung Kim, and Michael J. Schulte. 2012. The case for GPGPU spatial multitasking. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA '12)*. 1–12. https://doi.org/10.1109/HPCA.2012.6168946

[2] Paramvir Bahl, Ranveer Chandra, Thomas Moscibroda, Rohan Murty, and Matt Welsh. 2009. White space networking with wi-fi like connectivity. In *ACM SIGCOMM Computer Communication Review (SIGCOMM '09)*, Vol. 39. 27–38. https://doi.org/10.1145/1594977.1592573

[3] Michela Becchi, Kittisak Sajjapongse, Ian Graves, Adam Procter, Vignesh Ravi, and Srimat Chakradhar. 2012. A virtual memory based runtime to support multi-tenancy in clusters with GPUs. In *21st international symposium on High-Performance Parallel and Distributed Computing (HPDC'12)*. Delft, The Netherlands, 97–108. https://doi.org/10.1145/2287076.2287090

[4] Andrea Di Biagio, Alessandro Barenghi, Giovanni Agosta, and Gerardo Pelosi. 2009. Design of a Parallel AES for Graphic Hardware using the CUDA framework. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*. IEEE, Rome Italy. https://doi.org/0.1109/IPDPS.2009.5161242

[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC '09)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[6] Jie Chen and Guru Venkataramani. 2014. CC-Hunter: Uncovering Covert Timing Channels on Shared Processor Hardware. In *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*. IEEE, Cambridge UK, 216–228. https://doi.org/10.1109/MICRO.2014.42

[7] Renan Correa Detomini, Renata Spolon Lobato, Roberta Spolon, and Marcos Antonio Cavenaghi. 2011. Using GPU to exploit parallelism on cryptography. In *6th Iberian Conference on Information Systems and Technologies (CISTI'11)*. IEEE, Chaves Portugal. http://ieeexplore.ieee.org/document/5974171

[8] Khaled M. Diab, M. Mustafa Rafique, and Mohamed Hefeeda. 2013. Dynamic Sharing of GPUs in Cloud Systems. In *IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*. May, IEEE, Cambride, MA, USA, 947–954. https://doi.org/10.1109/IPDPSW.2013.102

[9] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization* 8, 4 (2012). https://doi.org/10.1145/2086696.2086714

[10] William Enck, Peter Gilbert, Seungyeop Han, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 5 (2014). https://doi.org/10.1145/2619091

[11] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Understanding and mitigating covert channels through branch predictors. *ACM Transactions on Architecture and Code Optimization* 13, 1 (2016), 10. hhttp://dx.doi.org/10.1145/2870636

[12] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'16)*. June, 279–299. https://doi.org/10.1007/978-3-319-40667-1_14

[13] Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanath, and Mohit Tiwari. 2015. Understanding contention-based channels and using them for defense. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*. IEEE, Burlingame CA USA, 639–650. https://doi.org/10.1109/HPCA.2015.7056069

[14] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. 2016. A complete key recovery timing attack on a GPU. In *IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. IEEE, Barcelona Spain, 394–405. https://doi.org/10.1109/HPCA.2016.7446081

[15] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. 2017. A Novel Side-Channel Timing Attack on GPUs. In *Proceedings of the on Great Lakes Symposium on VLSI (VLSI'17)*. 167–172. https://doi.org/10.1145/3060403.3060462

[16] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. 2014. Stealing Webpage Rendered on your Browser by Exploiting GPU Vulnerabilities. In *IEEE Symposium on Security and Privacy (SPI'14)*. IEEE, San Jose CA USA, 19–33. https://doi.org/10.1109/SP.2014.9

[17] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. 2016. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. Barcelona, Spain, 406–418. https://doi.org/10.1109/HPCA.2016.7446082

[18] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy (SP'15)*. IEEE, San Jose, CA, USA. https://doi.org/10.1109/SP.2015.43

[19] Chao Luo, Yunsi Fei, Pei Luo, Saoni Mukherjee, and David Kaeli. 2015. Side-Channel Power Analysis of a GPU AES Implementation. In *33rd IEEE International Conference on Computer Design (ICCD'15)*. https://doi.org/10.1109/ICCD.2015.7357115

[20] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *39th Annual International Symposium on Computer Architecture (ISCA'12)*. Portland, OR, USA, 118–129. https://doi.org/10.1109/ISCA.2012.6237011

[21] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. 2015. Thermal covert channels on multi-core platforms. In *24th USENIX Security Symposium*. Washington, D.C., 865–880. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/masti

[22] Clémentine Maurice, Christoph Neumann, Olivier Heen, and AurÃl'lien Francillon. 2014. Confidentiality Issues on a GPU in a Virtualized Environment. In *International Conference on Financial Cryptography and Data Security*. 119–135.

[23] Clémentine Maurice, Manuel Weber, Micheal Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. 2017. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *Network and Distributed System Security Symposium (NDSS'17)*. January. https://doi.org/10.14722/ndss.2017.23294

[24] Wen mei Hwu. 2011. *GPU Computing Gems* (1st. ed.). Elsevier.

[25] Naoki Nishikawa, Keisuke Iwai, and Takakazu Kurokawa. 2011. High-performance symmetric block ciphers on CUDA. In *Second International Conference on Networking and Computing (ICNC'11)*. Osaka Japan, 221–227. https://doi.org/10.1109/ICNC.2011.40

[26] NVIDIA. 2017. GPU Cloud Computing. (2017). Retrieved August 20, 2017 from http://www.nvidia.com/object/gpu-cloud-computing.html

[27] NVIDIA. 2017. Multi-Process Service. (2017). Retrieved March 2017 from https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

[28] NVIDIA. 2017. The Vulkan API. (2017). Retrieved August 20, 2017 from https://developer.nvidia.com/Vulkan

[29] Lena E. Olson, Jason Power, Mark D. Hill, and David A. Wood. 2015. Border Control: Sandboxing Accelerators. In *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. Waikiki HI USA, 470–481. https://doi.org/10.1145/2830772.2830819

[30] Lena E. Olson, Simha Sethumadhavan, and Mark D. Hill. 2015. Security Implication of Third-Party Accelerator. *IEEE Computer Architecture Letters* 15, 1 (2015), 50–53. https://doi.org/10.1109/LCA.2015.2445337

[31] Antonio J Peña, Carlos Reaño, Federico Silla, Rafael Mayo, Enrique S Quintana-Ortí, and José Duato. 2014. A complete and efficient CUDA-sharing solution for HPC clusters. *Parallel Comput.* 40, 10 (2014), 574–588. https://doi.org/10.1016/j.parco.2014.09.011

[32] Colin Percival. 2005. Cache missing for fun and profit. In *BSDCan*. https://doi.org/10.1.1.144.872

[33] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. 2016. CUDA leaks: Information Leakage in GPU Architecture. *ACM Transactions on Embedded Computing Systems (TECS)* 15, 1 (2016). https://doi.org/10.1145/2801153

[34] Vignesh T. Ravi, Michela Becchi, Gagan Agrawal, and Srimat Chakradhar. 2011. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In *20th international symposium on High performance distributed computing (HPDC'11)*. San Jose, CA, USA, 217–228. https://doi.org/10.1145/1996130.1996160

[35] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proc. ACM conference on Computer and communications security (CCS'09)*. Chicago, Illinois, USA, 199–212. https://doi.org/10.1145/1653662.1653687

[36] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. 2014. Enabling preemptive multiprogramming on GPUs. In *41st annual international symposium on Computer architecuture (ISCA'14)*. Minneapolis, Minnesota, USA, 193–204. http://dl.acm.org/citation.cfm?id=2665702

[37] Giorgos Vasiliadis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. Pixelvault: Using gpus for securing cryptographic operations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. Scottsdale Arizona USA, 1131–1142. https://doi.org/10.1145/2660267.2660316

[38] Zhenghong Wang and Ruby B. Lee. 2006. Covert and Side Channels Due to Processor Architecture. In *22nd Annual Computer Security Applications Conference (ACSAC '06)*. IEEE, Miami Beach, FL, USA, 473–482. https://doi.org/10.1109/ACSAC.2006.20

[39] Zhenghong Wang and Ruby B Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. In *34th annual international symposium on Computer architecture (ISCA'07)*. San Diego, CA, 494–505. https://doi.org/10.1145/1250662.1250723

[40] Zhenghong Wang and Ruby B. Lee. 2008. A novel cache architecture with enhanced performance and security. In *41st IEEE/ACM International Symposium on Microarchitecture (MICRO'08)*. IEEE, Lake Como Italy, 83–93. https://doi.org/10.1109/MICRO.2008.4771781

[41] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2015. Simultaneous Multikernel: Fine-grained Sharing of GPGPUs. *IEEE Computer Architecture Letters* 15, 2 (2015), 113–116. https://doi.org/10.1109/LCA.2015.2477405

[42] NVIDIA Whitepaper. 2012. VIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. (2012).

[43] Henry Wong, M. M. Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS'10)*. https://doi.org/10.1109/ISPASS.2010.5452013

[44] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. 2016. Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming. In *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. IEEE, Seoul South Korea. https://doi.org/10.1109/ISCA.2016.29

[45] Mengjia Yan, Yasser Shalabi, and Josep Tolrrellas. 2016. ReplayConfusion: Detecting Cache-based Covert Channel Attacks Using Record and Replay. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, Taipei Taiwan. https://doi.org/10.1109/MICRO.2016.7783742

[46] Zhiting Zhu, Sangman Kim, Yuri Rozhanski, Yige Hu, Emmett Witchel, and Mark Silberstein. 2017. Understanding the Security of Discrete GPUs. In *Proceedings of the General Purpose GPUs (GPGPU'10)*. Austin TX USA, 1–11. https://doi.org/0.1145/3038228.3038233