# CUDA Leaks: A Detailed Hack for CUDA and a (Partial) Fix

ROBERTO DI PIETRO, Bell Labs, France
FLAVIO LOMBARDI, Istituto per le Applicazioni del Calcolo, IAC-CNR, Rome, Italy
ANTONIO VILLANI, Roma Tre University, Rome, Italy

Graphics processing units (GPUs) are increasingly common on desktops, servers, and embedded platforms. In this article, we report on new security issues related to CUDA, which is the most widespread platform for GPU computing. In particular, details and proofs-of-concept are provided about novel vulnerabilities to which CUDA architectures are subject. We show how such vulnerabilities can be exploited to cause severe information leakage. As a case study, we experimentally show how to exploit one of these vulnerabilities on a GPU implementation of the AES encryption algorithm. Finally, we also suggest software patches and alternative approaches to tackle the presented vulnerabilities.

CCS Concepts: • **Security and privacy** → **Systems security;** *Operating systems security*; • **Computing methodologies** → *Parallel programming languages*; • **Software and its engineering** → Source code generation

Additional Key Words and Phrases: GPU, GPGPU, information leakage, registers

**ACM Reference Format:**
Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. 2016. CUDA leaks: A detailed hack for CUDA and a (partial) fix. ACM Trans. Embed. Comput. Syst. 15, 1, Article 15 (January 2016), 25 pages.
DOI: http://dx.doi.org/10.1145/2801153

## 1. INTRODUCTION

Graphics processing units (GPUs) are multicore processing units originally developed to accelerate graphics rendering. Today, they are also used to speed up general-purpose computation. As an example, GPUs are used in scientific compute-intensive tasks and in computational finance operations [Gaikwad and Toke 2010]. Multicores are also an effective solution supporting the performance requirements of real-time embedded systems [Kim et al. 2013; Paolieri et al. 2013; Aaraj et al. 2011]. GPUs have also been used to offload the CPU from security-sensitive tasks [Lombardi and Di Pietro 2010]. Further, several implementations of the most widespread cryptographic algorithms are now available on GPUs [Di Biagio et al. 2009; Vasiliadis et al. 2014; Nishikawa et al. 2011].

This work has been partially supported by the European Antitrust Forensic IT Tools project (rif. HOME/2012/ISEC/FP/C2/4000003977) funded by the Prevention of and Fight against Crime Programme of the European Union European Commission—Directorate—General Home Affairs.

Authors' addresses: R. Di Pietro, Cyber Security Research Department, Bell Labs, Route de Villarceaux, 91620 Nozay, France; email: roberto.di-pietro@alcatel-lucent.com; F. Lombardi, IAC-CNR, Istituto per le Applicazioni del Calcolo, via dei Taurini 19, 00185 Roma, Italy; email: flavio.lombardi@cnr.it; A. Villani, Maths and Physics Department, Roma Tre University, Largo San Leonardo Murialdo 1, 00159 Rome Italy; email: villani@mat.uniroma3.it. R. Di Pietro is also with the Maths Department of the University of Padova; email: dipietro@math.unipd.it.

DOI: http://dx.doi.org/10.1145/2801153

Indeed, some of the most powerful computer clusters in the world are based on GPUs (e.g., Dongarra et al. [1993]). Furthermore, cloud providers offer specialized services designed to deliver the power of GPU processing in the cloud [Zillians 2012], allowing multiple GPUs to be shared across different customers (i.e., GPU-as-a-Service). In addition, as single GPUs become more powerful, it is increasingly convenient to share even a single one among different tenants.

A different scenario is the one related to present PCs where the GPU is used for both graphical and computational tasks. As an example, modern browsers (e.g., Google Chrome) can use the GPU to render Web content; the same GPU can also be used to execute general-purpose computations [Lee et al. 2014]. The same is true for embedded devices, in which GPUs can execute arbitrary parallel code [Kim et al. 2013].

Despite being widespread, a thorough analysis of the GPU environment from a security point of view is lacking. In fact, as will be shown in the following, GPU and CPU architectures are quite different, and therefore they are subject to different threats. Running a task on a GPU requires performing three main steps: (i) a host application (i.e., a regular application running on the CPU) requests the execution of a *kernel* (i.e., a code to be run in parallel on the GPU), (ii) the host application copies the input data from host memory onto GPU memory, and (iii) the host application launches the kernel and gets back the results. Data transfers from the host application to the GPU are performed via a proprietary device driver: once data enters GPU memory, the device driver takes control. Therefore, the isolation between different kernels' data is mainly a responsibility of the GPU device driver. Since GPU memory stores a copy of the process-specific data, a flaw in the isolation mechanisms on the GPU would undermine the isolation mechanisms of the Operating System (OS), thus causing information leakage.

Any kind of information leakage from security-sensitive applications (e.g., encryption algorithms) would seriously hurt the trustworthiness of the shared-GPU computing model, where the term *shared GPU* indicates all of those scenarios where a GPU resource is actually shared among different users (or applications), be it on a server, on a mobile device or on a GPU cloud. The security implications on both GPU computing clusters and on remote GPU-as-a-Service offerings, such as those by companies like SoftLayer and Amazon, can be dramatic. It is worth mentioning that the findings of our work are not confined to the cloud environment; in present PCs (i.e., where the same GPU is used for both graphical and computational tasks), a malicious CUDA program would be able to *see* the content rendered by the GPU and thus violate the privacy of the user. Even worse, it would remain completely undetected since current antivirus software cannot analyze GPU binary code.

Due to its sensitiveness, one would expect the existence of secure and robust memory protection mechanisms on the GPU. Unfortunately, current GPUs and their device drivers are aimed at performance rather than security and isolation. As a consequence, GPU architectures are not robust enough when it comes to security [Larabel 2012; Kindratenko et al. 2009], and the adoption of GPUs actually introduces new threats that require specific considerations. Further, in view of the GPU virtualization approach offered by the upcoming hypervisors, information leakage risks would even increase. In fact, there is very limited hardware support for virtualization technology in commonly available GPUs. A first attempt to virtualize GPUs has recently been introduced by NVIDIA with GRID boards [NVIDIA 2014b]. However, virtualized GPU security is outside the scope of this work.

*Contribution*. This article provides several contributions to the novel problem of secure computing on GPUs, with a focus on the CUDA platform. In detail, leveraging perfectly standard GPU code, we were able to produce information leakage flaws; we were able to cause leakages by stressing the existing CUDA memory allocation and deallocation primitives, which led to the discovery of three critical vulnerabilities. As

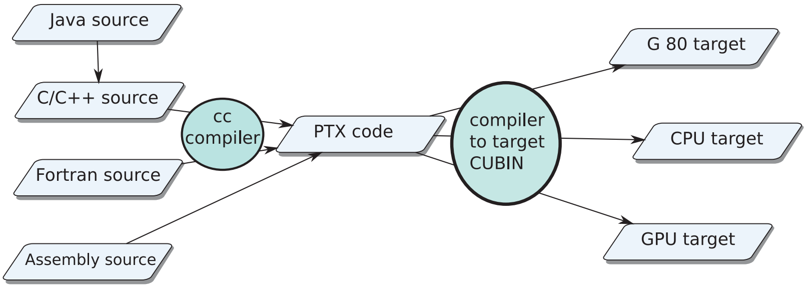


Fig. 1. Main steps in compiling source code for CUDA devices.

for the first vulnerability described in this work, we were able to induce information leakage on GPU shared memory. Further, an information leakage vulnerability based on GPU global memory and another one based on GPU register spilling over global memory were discovered and discussed.

As a case study, we evaluated the impact of one of these leakages on a publicly available GPU implementation of the AES. In particular, we demonstrated that through the global memory vulnerability, it is possible, for a not-legitimate user, to access both the plaintext and the encryption key. Finally, we propose and discuss countermeasures and alternative approaches to fix the highlighted vulnerabilities.

## 2. CUDA ARCHITECTURE

CUDA is a parallel computing platform for NVIDIA GPUs. In particular, CUDA is composed of three parts: the device ddriver, the runtime, and the compilation toolchain. The device driver handles the low-level interactions with the GPU (e.g., task scheduling); the runtime handles the requests coming from CUDA applications (e.g., dynamic memory allocation) and routes such requests to the device driver. The compilation toolchain allows compilation of CUDA applications from source code into intermediate and executable binary code [Papakonstantinou et al. 2013].

A CUDA application is composed of host code (running on the CPU) and one or more kernels (running on the GPU). Kernels are special functions that are executed $N$ times in parallel by $N$ different CUDA threads (i.e., threads running on the GPU). Once a kernel is scheduled on the GPU, it always runs until completion, and there is no clean way for the OS to stop its execution. Only the device driver can interrupt a running kernel by launching a specific interrupt.

The compilation of a CUDA application is performed in two steps: (i) the compilation toolchain transforms the kernel source code into an intermediate language called *PTX* [NVIDIA 2014a] , and (ii) the device driver translates the PTX into a binary code called *CUBIN* (CUda BINary), which is tailored to the specific GPU where it will be executed. This approach allows specific code optimization to be tied to the actual GPU resources. An overview of the CUDA compilation process is depicted in Figure 1.

The CUDA architecture can be synthesized as follows: (i) a binary file consisting of the host and PTX [NVIDIA 2014a] object code; (ii) the CUDA user-space closed-source library (libcuda.so); (iii) the NVIDIA kernel-space closed-source GPU driver (nvidia.ko); and (iv) the hardware GPU with its interconnecting bus (PCI Express or PCIe), memory (global, shared, local, registers), and computing cores (organized in blocks and threads).

The runtime offers a handle-based, imperative API: most objects are referenced by opaque handles that may be passed to functions to manipulate the objects. For our purposes, the *cudaContext* is the most important handle in the runtime. CUDA applications use *cudaContexts* to cope with relevant tasks such as virtual memory management

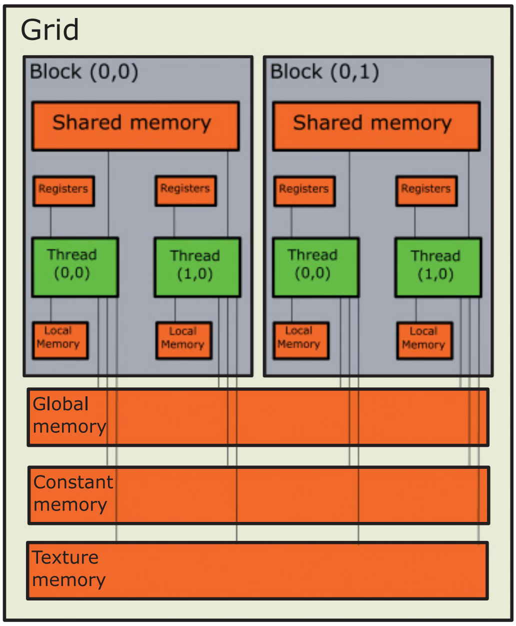


Fig. 2. CUDA memory hierarchy and threads.

for both host and GPU memory. The device driver is responsible for allocating and managing resources belonging to a *cudaContext* as well as for freeing up these resources when a *cudaContext* is disposed of. It is worth noting that a *cudaContext* is automatically disposed of during the host process termination, or as an alternative, it can be cleaned up with a call to a specific function provided by the runtime (i.e., *cuCtxDestroy*).

### 2.1. CUDA Memory Hierarchies

CUDA features different memory spaces and types (e.g., global memory, shared memory). Main memory layers are depicted in Figure 2. Please note that Figure 2 shows only a logical organization of the CUDA memory hierarchy. For instance, depending on the size of the reserved memory, the compiler may choose to map local memory on registers or on global memory. All threads have access to the same global memory. Each CUDA thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. CUDA specifications do not describe what happens to shared memory when a block completes its execution. There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are persistent across kernel launches by the same application [NVIDIA 2014a]. Considering that this work focuses on global memory, shared memory, and registers, they are detailed in the following sections.

*2.1.1. Global Memory.* Global memory is accessed via 32-, 64-, or 128-byte memory transactions. This is by far the largest type of memory available inside the GPU. When a warp (i.e., the minimum number of threads that can be scheduled on the GPU) executes an instruction that accesses global memory, it coalesces [Jang et al. 2011b] the memory accesses of the threads within the warp into one or more memory transactions. This allows amortizing memory access latency.

*2.1.2. Shared Memory.* Shared memory is faster than global memory and is located near each processor core in order to have low-latency access (similarly to cache memory).

Each multiprocessor (i.e., fixed group of cores) is equipped with the same amount of shared memory. The size of the shared memory is in the order of kilobytes (e.g., 16KB or 64KB times the number of the available multiprocessors). Thanks to shared memory, threads belonging to the same block can efficiently cooperate by seamlessly sharing data. The information stored inside a shared memory bank can be accessed only by threads belonging to the same block. Each block can be scheduled onto one multiprocessor per time. As such, a thread can only access the shared memory available to a single multiprocessor. The CUDA developers guide [NVIDIA 2014a] encourages coders to make use of this memory as much as possible. In particular, specific access patterns are to be followed to reach maximum throughput. Shared memory is split into equally sized memory modules called *banks*, which can be accessed simultaneously.

*2.1.3. Registers.* CUDA registers represent the fastest and smallest latency memory of GPUs. However, as we will show later, CUDA registers are prone to leakage vulnerabilities. The number of registers used by a kernel can have a significant impact on the number of resident warps: the fewer registers a kernel uses, the more threads and thread blocks are likely to reside on a multiprocessor, improving performance. Therefore, the compiler uses heuristics to minimize register usage through register spilling [Micikevicius 2011]. This mechanism places variables (that could have exceeded the number of available registers) in local memory.

### 2.2. Preliminary Considerations

The CUDA programming model assumes that both the host and the device maintain their own *separate memory spaces* in DRAM, respectively host memory and device memory. Therefore, a program manages the global, constant, and texture memory spaces through calls to the CUDA runtime. This includes device memory allocation and deallocation as well as data transfer between host and device memory. Such primitives implement some form of memory protection that is worth investigating. In fact, it would be interesting to explore the possibility of accessing these memory areas bypassing such primitives. Moreover, this would imply analyzing what kind of memory isolation is actually implemented. In particular, it would be interesting to investigate whether it is possible to obtain a specific global memory location by leveraging GPU allocation primitives. Further, when two different *kernels* $A$, $B$ are being executed on the same GPU, it would be interesting to know what memory addresses can be accessed by *kernel* $A$ or if $A$ can read or write locations that have been allocated to $B$ in global memory. What happens to memory once it is deallocated is undefined, and released memory is not guaranteed to be zeroized [Evans et al. 1994; Van Tilborg and Jajodia 2011; Henson and Taylor 2014].

Finally, note that the trend in memory hierarchy is to have a single unified address space between GPU and CPU (see also NVIDIA [2014a]). In fact, the CUDA unified virtual address space mode (unified virtual addressing) puts both CPU and GPU execution in the same address space. This alleviates CUDA software from copying data structures between address spaces, but it can be an issue on the driver side since GPUs and CPUs can compete over the same memory resources. In fact, such a unified address space can allow potential information leakage.

## 3. RATIONALES OF VULNERABILITIES RESEARCH

The strategy adopted by IT companies to preserve trade secrets mostly consists of hiding architectural and implementation details of their products. Although this is considered the best strategy from a commercial point of view, as for what concerns security, this approach usually leads to unexpected breaches [Mercuri and Neumann 2003]. The security-through-obscurity approach has been embraced by the graphics technology companies as well, and most implementation details about the CUDA architecture are not publicly available.

As detailed next, once a process invokes a CUDA routine, the process has to completely trust the NVIDIA implementation of both the driver and the runtime. However, if we only consider the public information about the architecture, it is unclear if any of the security mechanisms that are usually enforced in the OS are maintained inside the GPU. The only implementation details available via official sources just focus on performance. Indeed, NVIDIA suggests the use of specific access patterns to global memory to achieve the highest throughput. In contrast, important implementation details about security features are simply omitted. As an example, there is no official information on whether memory is zeroized after releasing it [Van Tilborg and Jajodia 2011]. Although this is not a problem for most scientific applications, it can cause severe security issues when sensitive data are involved in the computation. NVIDIA implemented memory isolation between different *cudaContext* within its closed-source driver. Such a choice can introduce vulnerabilities, as explained in the following example. Suppose that an host process $P_a$ has to perform some computation on a generic data structure $S$. The computation on $S$ must be offloaded to the GPU for performance reasons. Hence, $P_a$ allocates some host memory $M_h^a$ to store $S$, then it reserves memory on the device $M_d^a$ and copies $M_h^a$ to $M_d^a$ using the CUDA runtime primitives. From this moment onward, the access control on $M_d^a$ is not managed by the host OS and CPU. It becomes exclusive responsibility of the NVIDIA GPU driver—that is, the driver takes the place of the OS. Unfortunately, GPU drivers do not usually undergo the thorough security-focused review to which OS are usually subject.

To make things even worse, providing memory isolation in CUDA is probably far more complex than in traditional CPU architectures. As described in Section 2, CUDA threads may access data from multiple memory spaces during their execution. Although this separation allows one to improve the performance of the application, it also increases the complexity of access control mechanisms and makes it prone to security breaches. A solution that preserves isolation in memory spaces like global memory, which in recent boards reaches the size of several gigabytes, could be unsuitable for more constrained resources like shared memory or registers. Indeed, both shared memory and registers have peculiarities that rise the level of complexity for the memory isolation process. The shared memory, for example, is like a cache memory with the distinguishing feature that it is directly usable by the developers. This is in contrast to more traditional architectures, such as *x86*, where the behavior of the cache is transparent to software.

For what concerns registers, a feature that could taint memory isolation is that registers can be used to access global memory as well: in fact, a GPU feature (named *register spilling* [Micikevicius 2011]) allows one to map a large number of kernel variables onto a small number of registers. When the GPU runs out of hardware registers, it can transparently leverage global memory instead.

Multiple memory transfers across the PCIe bus for the global, constant, and texture memory spaces are costly. As such, they are made persistent across kernel launches by the same application. This implies that the NVIDIA driver stores application-related *state information* in its data structures. As a matter of fact, in case of interleaved execution of CUDA kernels belonging to different host processes, the driver should prevent any process $P_j$ to perform unauthorized access to memory locations reserved to any other process $P_i$. Hence, this architecture raises questions as to whether it is possible for a process $P_j$ to obtain unauthorized access to the GPU memory of any other process $P_i$. Our working hypothesis, as for the strategy adopted by GPU manufacturers, is that they lean to trade off security with performance. Indeed, one of the main objectives of the general-purpose GPU (GPGPU [Buck et al. 2004])

Table I. Summary of the Results of the Experiments

| | Leakage | Preconditions |
|---|---|---|
| Shared memory | Complete | $P_a$ is running |
| Global memory | Complete | $P_a$ has terminated and $P_b$ allocates the same amount of memory as $P_a$ |
| Registers | Partial | None |

computing framework is to speed up computations in high-performance computing and not to provide context isolation. In such a scenario, the overhead introduced by a memory initialization routine run after each kernel would introduce an unacceptable overhead for the GPU standard [Yang et al. 2011]. Indeed, as it will be proved and detailed in Section 4, these mechanisms have severe flaws and leak information.

It is worth mentioning that providing a thorough explanation of the root causes of the discovered vulnerabilities is outside the scope of this work. Our aim is to demonstrate that GPUs can leak information and to provide hints for countermeasures. Furthermore, even if GPUs are not designed with security in mind, they are increasingly used to access sensitive data. As such, it is important to investigate how GPU security can be improved to provide better security guarantees.

## 4. EXPERIMENTAL RESULTS

The performed experiments aim at discovering whether, and under which conditions, a violation of the memory isolation mechanisms could occur—that is, whether the memory belonging to an honest process $P_a$ can be accessed by a malicious process $P_b$.

This section describes the test campaign we set up on GPU hardware to investigate the issues mentioned previously. We performed the experiments on two different generations of CUDA-enabled devices (Fermi and Kepler) using a black-box approach. It is important to note that in our experiments, we consider an adversary that is able to interact with the GPU only through legitimate invocations to CUDA runtime.

In the following sections, we will detail three different leakage attacks targeted at different memory spaces. Each leakage has specific preconditions and characteristics. For each kind of leakage, we developed a C program making use of the standard CUDA Runtime Library. In just a single case, we had to directly write PTX assembly code to obtain the desired behavior. The results of our experiments are summarized in Table I.

The rest of this section is organized as follows: the experimental testbed is described in detail in Section 4.1. In Section 4.2, the first and simplest leakage is discussed regarding shared memory. In Section 4.3, a potentially much more extended (in size) information leakage is detailed—while not leveraging shared memory. Finally, the most complex and powerful information leakage is described in Section 4.4. It leverages registry usage and local memory.

### 4.1. Testbed Setup

Tests were performed on the Linux platform, as GPU clusters are mainly hosted on such OS. We performed the experiments using different CUDA HW/SW configurations.

The testbed features both COTS and professional-level CUDA hardware using production-level SDKs and was comprised of the HW/SW configurations described in Table II. To verify whether the obtained information leakage was independent from the implementation of a specific GPU, and thus to make our experiments more general, two radically different configurations were chosen—on the one hand, a Tesla card that can be considered targeting the HPC sector, and on the other hand, a GeForce card targeted at consumers and enthusiasts. The objective was not performance comparison but the analysis of possible leakages in different scenarios. The Tesla card implements

Table II. The Testbed Used for the Experiments

| GPU Model | Tesla C2050 | GeForce GT 640 |
|---|---|---|
| CUDA Driver | 4.2 | 5.0 |
| CUDA Runtime | 4.2 | 4.2 |
| CUDA Capability | 2.0 | 3.0 |
| GPU Architecture | Fermi GF100 | Kepler GK107 |
| Global Memory | 5GB | 2GB |
| Shared Memory per Multiprocessor | 48KB | 16KB |
| Registers | 32,768 | 65,536 |
| Multiprocessors | 14 | 2 |
| Total Shared Memory | 672KB | 32KB |

the Fermi architecture, whereas the GeForce card belongs to the newer Kepler family (i.e., the latest generation of NVIDIA GPUs). As such, the two GPUs differ with respect to the supported CUDA Capability (2.0 for the Fermi and 3.0 for the GeForce). In our experiments, the compiling process took into consideration the differences between the target architectures. In Table II , we report the specifications of the two GPUs. The reported size of shared memory and registers represent the amount of memory available for a single block (see NVIDIA [2014a]).

Each experiment was replicated on both configurations; in some cases, we tuned some of the parameters to explicitly fit the GPU specifications (e.g., the size of shared memory).

### 4.2. Shared Memory Leakage

In this scenario, the objective of the adversary is to read information stored in shared memory by another process. To do so, the adversary uses regular runtime API functions. Present CUDA runtime allows each *cudaContext* to have exclusive access to the GPU. As a consequence, CUDA runtime does not feature any preemption mechanism among different *cudaContexts*.

The runtime continues to accept requests even when there is a kernel running on the GPU. Such requests are in fact queued and later served according to a FIFO scheduling policy. It is worth noting that the preceding requests can belong to different *cudaContexts*. As such, information can be leaked if no memory cleaning functionality is invoked, following a context switch.

In fact, every time a malicious process is rescheduled on the GPU, it can potentially read the last-written data of the previous process that used the GPU. As such, the scheduling order affects which data is exposed. As an example, if $P_a$ is performing subsequent rounds of an algorithm, the state of the data that can be read reflects the state reached by the algorithm itself.

The experiment to validate such a hypothesis is set up as follows: two different host-threads belonging to distinct processes are created: $P_a$ being the honest process and $P_b$ the malicious one trying to sneak trough $P_a$ memory. $P_a$ executes $K$ times a kernel that writes in shared memory (the impact of the $K$ parameter is detailed next). In this experiment, $P_a$ copies a vector $V_g$.[1] Every element in $V_g$ is of type *uint32_t* as defined in the header file *stdint.h*. The size of the vector is set equal to the size of the physical shared memory: 48KB for the Tesla C2050 and 16KB for the GeForce GT 640. The copy proceeds from global memory to shared memory. The host thread initializes $V_g$ deterministically using sequential values (i.e., $V_g[i] = i$). $P_b$ executes $K$ invocations of a kernel that reads shared memory. In particular, $P_b$ allocates a vector $V_g$ and declares

[1] $V_g$ denotes any vector stored in global memory, whereas $V_s$ denotes any vector stored in shared memory.

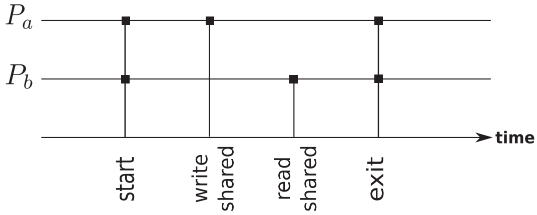


Fig. 3. The schedule that causes the leakage on shared memory.

in shared memory a vector $V_s$ with size equal to the shared memory size. Then data is copied from $V_s$ to $V_g$.

With this particular sequence of operations (Figure 3), $P_b$ recovers exactly the same set of values written by $P_a$ in shared memory during its execution. In other words, a complete information leakage of the memory used by $P_a$ occurs. However, the scheduling order of processes is nondeterministic and is decided by the OS. As such, to cause the leakage, it is essential for the kernel of $P_b$ to be scheduled on the GPU before the termination of host process $P_a$. In fact, we experimentally verified that shared memory is zeroized [Lee et al. 2005] by the CUDA runtime before the host process invokes the *exit()* function.

One of the parameters of the function developed for this experiment is the kernel block and grid size. In our tests, we have adopted a number of blocks equal to the actual number of physical multiprocessors of the GPU. With regard to the number of threads, we have specified a size equal to the warp size.

Quite surprisingly, the values captured by $P_b$ appear exactly in the same order as they were written by $P_a$. Given that GPUs have a block of shared memory for each multiprocessor, one would expect that a different scheduling of such multiprocessors would lead to different orderings of the read values. To investigate this behavior, we made another test using an additional vector, where the first thread of each thread block writes the identifier of the processor where it is running. This info is contained in the special-purpose register *smid* that the CUDA instruction set architecture (ISA) is allowed to read. In fact, *smid* is a predefined, read-only special register that returns the processor (SM) identifier on which a particular thread is executing. The SM identifier ranges from 0 to *nsmid* − 1, where *nsmid* represents the number of available processors. To read this information, we embedded the following instruction, written in PTX assembler, inside the kernel function:

```
asm("mov .u32 %0, \% smid;" : "=r" (ret) );
```

The preceding instruction copies the unsigned 32-bit value of register *%smid* into the *ret* variable residing in global memory. Note that CUDA deterministically chooses the multiprocessors for the first *nsmid* blocks. As an example, for the Fermi card, we obtained the multiprocessor ID sequence: 0, 4, 8, 12, 2, 6, 10, 13, 1, 5, 9, 3, 7, 11. This evidence explains the reason why the process $P_b$ was able to read the values in the same order as they were written by process $P_a$. Due to the closed-source nature of CUDA, we cannot claim that the multiprocessor ID sequence would be always the same. Indeed, by varying some configuration parameters (e.g., the number of blocks or the number of threads), a different scheduling of multiprocessors could be triggered. However, this behavior would impact only on the order of the leaked values.

*The impact of K on the shared memory leakage.* Figure 3 shows the scheduling (i.e., the sequence $S$ of operations) that causes the leakage. $S$ depends on the nondeterministic choices performed by the OS scheduler during the experiment. Even though the host

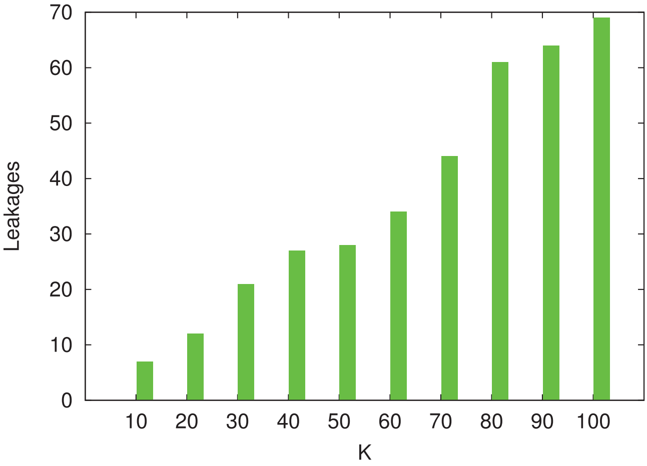


Fig. 4. The impact of the value of $K$ on the shared memory leakages.

was in idle state when we ran the experiments, other processes such as kernel threads and OS daemons were running. As such, they could have been scheduled on the same CPU core as $P_a$ and $P_b$. This may result in a different sequence of operations ($S$'), which does not cause the leakage. To better clarify this phenomenon, Figure 4 shows the impact of $K$ on the number of leakages. $K$ varies from 10 to 100 with step 10. Each bin in Figure 4 represents the number of leakages for the corresponding value of $K$. As an example, with $K = 100$, $S$ occurred 69 times, whereas with $K = 10$, the leakage occurred 7 times. Please note that the average success rate for such an attack was always around 70% in our tests. The reported values have been estimated by calculating the average value over 128 experiments. Surprisingly, increasing the number of runs does not increase the chances for $S$ to occur (indeed, only the absolute value of leakages grows), because to obtain a fair result, we run all experiments in the same conditions (i.e., in idle state). Different conditions may impact on the success rate of $S$ to occur. However, exploring the most profitable conditions from an adversary's perspective is left as future work.

*Take-away point*. GPU shared memory can leak information by leveraging interleaved access by different host programs.

### 4.3. Global Memory Leakage

In this scenario, the objective of the adversary is to read information stored in GPU global memory by another process. As in the previous scenario, we have two independent host processes, namely $P_a$ and $P_b$, representing the honest and the malicious process, respectively. The information leakage is due to the lack of memory-zeroizing operations. In fact, as mentioned in Section 2, some of the resources used by a *cudaContext* are cleaned up only when the context is destroyed.

In this experiment (Figure 5), $P_a$ is executed first and allocates four vectors $V_1, V_2, V_3, V_4$ of $D$ bytes each in GPU memory. The dynamic allocation on the GPU uses the *cudaMalloc()* primitive of the CUDA runtime. The $i$-th elements of vectors $V_1$ and $V_2$ are initialized as follows:

$$V_1[i] = i; V_2[i] = D + i.$$

In this experiment, $P_a$ invokes a *kernel* that copies $V_1$ and $V_2$ into $V_3$ and $V_4$, respectively. $P_a$ then terminates, and $P_b$ gets scheduled. The correct synchronization

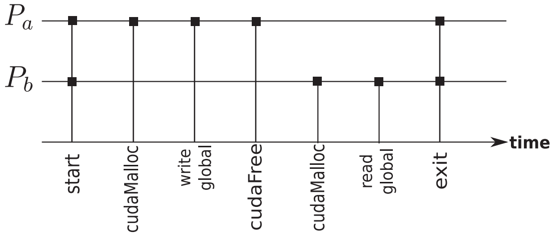


Fig. 5. The schedule that causes the leakage on global memory.

between the two processes is maintained through Unix sockets. $P_b$ allocates four vectors $V_1, V_2, V_3, V_4$ of size $D$ bytes each, just as $P_a$ did before.

The only difference is that $P_b$ does not initialize neither $V_1$ nor $V_2$. $P_b$ now runs the same kernel code that $P_a$ executed before and copies $V_3$ and $V_4$ back in the host memory.

We verified that $P_b$ obtains exactly the same content written before by $P_a$. This leakage is deterministic. Hence, we can conclude that the information leakage on the global memory is *full* (i.e., $P_b$ retrieved all data written by $P_a$ in global memory during its execution).

We tried several values of $D$ (starting from 4KB up to the maximum allocable memory), always obtaining a full leakage. However, we noticed that the leakage is full only if the malicious process allocates exactly the same amount of memory released by the honest one. This behavior could depend on the fact that the NVIDIA driver implements a modified buddy-system memory manager using different queues for memory requests of different size. Unfortunately, due to the closed-source nature of the NVIDIA driver, we were not able to verify this hypothesis.

*Take-away point*. GPU global memory can easily leak information by exploiting the lack of adequate data allocation and data cleaning mechanisms.

### 4.4. Register-Based Leakage

In this last described leakage, the objective of the adversary is to exploit registers to leak information. Again, we have two independent host processes, namely $P_a$ and $P_b$, representing the honest and the malicious process, respectively. $P_b$ exploits a feature called *register spilling* [Micikevicius 2011] to access the global memory reserved to $P_a$. Register spilling allows a process to reserve more registers then those actually available on the GPU. If a variable cannot be assigned to a register, then it is placed in global memory. By using the PTX intermediate language (see Figure 1), we can specify the exact number of registers that a kernel needs during its execution. If the required number of registers exceeds those that are physically available on chip, the compiler (PTX to CUBIN) starts using global memory instead of registers to store variables. The number of available registers per each block depends on the GPU capability (i.e., 32K for CUDA capability 2.0 and 64K for CUDA capability 3.0).

From point of view of the adversary, register spilling is an easy way to access global memory, bypassing runtime access primitives. In our experiments, we tried to understand whether register spilling is subject to the same access control of memory allocation primitives (e.g., *cudaMalloc()*). Surprisingly, we discovered that the malicious process can effectively exploit register spilling to access memory areas that had been reserved to other *cudaContexts*. Further, the malicious process can access such locations even while the legitimate process still owns them (namely before it calls the *cudaFree()*). This is why we believe that this latter leakage is the most dangerous among the presented ones. We were able to replicate such an attack only on the Kepler

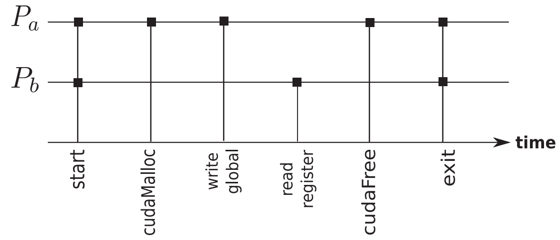


Fig. 6. The schedule that causes the leakage on registers. In this scenario, $P_b$ accesses the global memory without runtime primitives.

GPU, whereas the Fermi GPU seems immune from this attack. In the remainder of this section, we describe in detail the steps we followed to obtain the register-based leakage. As a preliminary step, we zeroized the whole memory available on the device to avoid tainting the results. As for previous experiments, we needed two independent host processes, $P_a$ and $P_b$, as the honest and the malicious processes, respectively.

$P_a$ performs several writes into the global memory, whereas $P_b$ tries to read the memory assigned by the driver to process $P_a$. To verify the outcome of the attack in a more reliable way, $P_a$ writes an easily recognizable pattern. In particular, $P_a$ allocates an array and marks the first location with the hexadecimal value *0xdeadbeef*. At position $j$ of the array, $P_a$ stores the value *0xdeadbeef* + $j$, where $j$ represents the offset in the array.

$P_b$ reserves a predefined number of registers and copies the content of the registers back to host memory; in this way, $P_b$ tries to exploit the register spilling to violate the memory locations reserved by $P_a$. Indeed, if the register spilling mechanism is not properly implemented, then some memory locations reserved to $P_a$ could be inadvertently assigned to $P_b$. We ran $P_a$ and $P_b$ concurrently and checked for any leaked location previously written by $P_a$.

As per Figure 6, in this case, $P_b$ succeeds in accessing memory locations reserved by $P_a$ before this latter executes the *cudaFree* memory releasing operation. Note that this behavior is different from the one observed for the global memory attack described in Section 4.3.

Algorithm 1 shows the pseudocode of the attacking process $P_b$. The array `dra` is used to save the values read through the register spilling, whereas `hra` is used to copy the content of `dra` into the host's memory. The size of the arrays is MAXNUMREGS*gridsize*blocksize (line 8). When the GPU kernel is invoked (line 13), each CUDA thread tries to steal MAXNUMREGS memory locations through the code represented in Figure 7. The function `get_reg 32bit()` in Figure 7 shows a PTX code fragment that can be used by the kernel for register reservation. In this case, $8,192$ registers are reserved. When the kernel `k_read_regs` completes its execution, the read values are copied back to host memory (line 14). The function `search_leaks_from_patterns()` (line 15) searches for known patterns in `hra` and returns the leaked locations (duplicates are removed).

Given that our Kepler GPU has two multiprocessors and a warp size of 32, we ran both $P_a$ and $P_b$ with a *gridsize* of two blocks and a *blocksize* of 32 threads. This way, all multiprocessors were occupied during tests to allow a more realistic leakage detection. In Table III, the number of bytes of $P_a$ that are read by process $P_b$ is reported. The analysis was conducted by varying the amount of registers declared by $P_b$ and by varying the amount of memory locations declared by $P_a$. Results show two rounds of the experiment. As an example, if $P_b$ reserves a 32KB register space (corresponding to 8K 32-bit registers) and $P_a$ allocates an amount of memory equal to 32 MB, by executing

Table III. Number of Bytes Leaked with Two Rounds of the Register Spilling Exploit

| | | Register Space Allocated by $P_b$ | | |
|---|---|---|---|---|
| | | **32KB** | **64KB** | **128KB** |
| Memory allocated by $P_a$ | **16MB** | 32K | 128K | 256K |
| | **32MB** | 64K | 128K | 256K |
| | **64MB** | 64K | 64K | 128K |
| | **128MB** | 64K | 64K | 256K |

**ALGORITHM 1:** The pseudocode that allows $P_b$ to access global memory through register spilling (only one round is represented)

```
Input:
$\vec{dra}$: The array in the device memory
MAXNUMREGS: The number of registers to spill
__global__ k_read_regs(dra)
begin
    thread_start_offset ← thread_global_id × MAXNUMREGS
    get_reg32bit(dra + thread_start_offset) /* Read from registers and copy in dra*/
end

Input:
$dra$: The array in the device memory
$gridsize$: The number of blocks
$blocksize$: The number of threads per block
$hra$: The array in the host memory
MAXNUMREGS: The number of registers to spill
Output: The leaked locations
FindLeakage($dra$,$hra$,$gridsize$,$blocksize$,MAXNUMREGS)
begin
    len=MAXNUMREGS ×$gridsize$ × $blocksize$
    dra ← cudamalloc(len)
    hra ← malloc(len)
    cudamemset(dra,0,l) /*zeroizing*/
    memset(dra,0,len)/*zeroizing*/
    k_read_regs <<<gridsize,blocksize >>>(dra, MAXNUMREGS) /* The kernel invocation
    */
    cudaMemcpy(hra, dra,len, cudaMemcpyDeviceToHost)
    return search_leaks_from_patterns(hra)
end
```

the mentioned experiment twice, we obtain an information leakage of 64KB (i.e., 16K 32-bit words), because in different rounds, the leakage consists of different memory locations. The rationale is that the dynamic memory management mechanism that is implemented in the GPU driver and in the CUDA runtime behaves as shown. As a consequence, this attack is even more dreadful, as the adversary (by executing several rounds) can potentially read the whole memory segment allocated by $P_a$.

*Take-away point*. GPU registers can leak information, since register allocation uses GPU memory when hardware registers are exhausted.

In our tests, the register-based attack always managed to produce the leakage. However, to better quantitatively evaluate this phenomenon, we investigated and analyzed

```
__device__
void get_reg32bit(uint32_t *regs32) {

      # declaration of 8300 registers
      asm(".reg .u32 r<8300>;\n\t");
      # move the content of register r0 into
      # the position 0 of regs32[]
      asm("mov.u32 %0, r0;" : "=r"(regs32[0]));
      asm("mov.u32 %0, r1;" : "=r"(regs32[1]));
      ...
      asm("mov.u32 %0, r8191;" : "=r"(regs32[8191]));

}
```

Fig. 7. A snippet of the code that allows to access global memory without *cudaMalloc*.

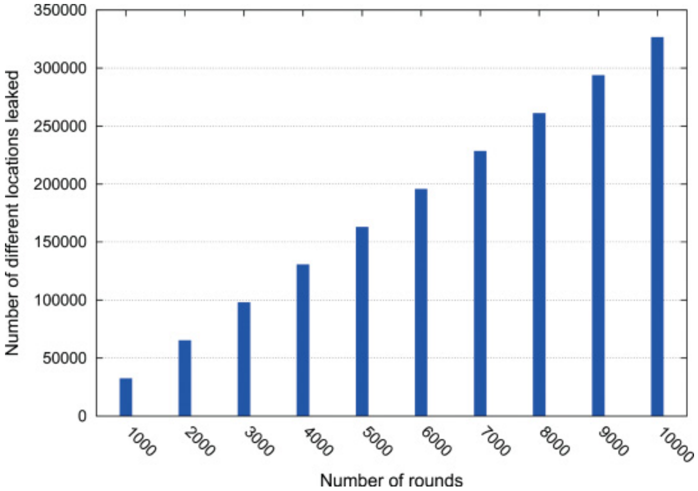


Fig. 8. The number of different locations leaked is proportional to the number of executed rounds.

the relationship between the number of locations where the leakage succeeds and the number of executed rounds.

In Figure 8, results are shown with respect to a number of rounds ranging from 1,000 to 10,000. Growth is linear in the number of rounds. In particular, the leakage starts from 32K locations for 1,000 rounds and reaches 320K locations leaked when the number of rounds is 10,000. The leaked locations belong to contiguous memory areas; the distance between each location is 32 bytes. For example, if the leaked locations start from byte 0 and end at byte 320, then we obtain 10 locations: one location every 32 bytes. We claim that this behavior depends on the implementation and on the configuration of the kernel. A more thorough analysis will be provided in future work.

The results of this experiment suggest a further study on the possibility for a malicious process to obtain write access to the leaked locations. To investigate this vulnerability, we performed an additional set of experiments.

We kept the same configuration as the previous test, but to foster the detection of the potential unauthorized write accesses, we used a cryptographic hash function. Indeed, thanks to the properties of hash functions, if the malicious process succeeds in interfering with the computation of the legitimate process—for instance, by altering even only a single bit—this would cause (with overwhelming probability) errors in the output of the legitimate process.

For $P_a$, we used a publicly available GPU implementation of the *SHA-1* hash function included into the *SSLShader*:[2] an SSL reverse proxy transparently translating SSL sessions to TCP sessions for back-end servers. The encryption/decryption is performed on-the-fly through a CUDA implementation of various cryptographic algorithms. Actually, the code implementing the GPU cryptographic algorithm is contained in the *libgpucrypto* library, which can be downloaded from the same Web site. For our experiments, we used version 0.1 of this library.

In this test, $P_a$ uses the GPU to compute the *SHA-1* $4,096$ times on a constant plaintext of 16KB. $P_a$ stores each hash in a different memory location. To test the integrity of GPU-computed hashes, $P_a$ also computes *SHA-1* on the CPU using the OpenSSL library and then compares this result with the ones computed on the GPU. The malicious process $P_b$ tries to taint the computation of $P_a$ by writing a constant value into the leaked locations.

In our test, we ran $P_a$ 1,000 times and concurrently launched the malicious process $P_b$. Even if in most cases we were able to read a portion of the memory reserved to $P_a$, the write instruction was ignored and all hashes computed on the GPU were correct.

*Take-away point*. The register spilling vulnerability does not seem to allow interfering with the computation of the legitimate process.

## 5. CASE STUDY: SSLSHADER

To evaluate the impact of the global memory vulnerability in a real-world scenario, we attacked the CUDA implementation of AES presented in Jang et al. [2011a] that is part of *SSLShader*. Such a scenario is becoming even more realistic, as GPUs are increasingly being used as network cryptographic accelerators [Bossuet et al. 2013; Verner et al. 2011]. Indeed, in this case, the encryption process can run for a very long time. As such, a single malicious unprivileged process would be able to perform the attack described in this case study. The SSLShader comes with some utilities that can be used to verify the correctness of the implemented algorithms. To run our experiments, we modified one of these utilities. In particular, we changed the AES test utility to encrypt a constant plaintext using a fixed key. We chose a constant plaintext of 4KB (i.e., the first two chapters of *Divine Comedy* written in LaTeX), and we set the 128-bit encryption key to the juxtaposition of the following words: *0xdeadbeef*, *0xcafed00d*, *0xbaddcafe*, and *0x8badf00d*.

In this experiment, we assume that the GPU is shared between the adversary and the legitimate process. Further, we assume that the adversary can read the ciphertext.

The steps performed by the attacking process are described in Algorithm 2. We consider the attack successful in two cases: in the first case, the adversary gets access to the *whole* plaintext (line 11)—achieving *plaintext leakage*; in the second case, the adversary obtains some words of the encryption key (line 8)—achieving *key leakage*. Even if this latter case can be less dreadful than the former one, it still jeopardizes security. Indeed, to obtain the desired information, the adversary could attack the undisclosed portion of the key (e.g., via brute force, differential cryptanalysis [Heys 2002]) and eventually decrypt the message.

The experiment is composed of the following steps. First, we run an infinite loop of the CUDA AES encryption; meanwhile, we ran Algorithm 2 100 times. To avoid counting a single leakage event more than once, each execution of Algorithm 2 zeroizes the memory (lines $9, 15, 19, 24$). We repeated this experiment 50 times on both the Kepler and the Fermi architectures, measuring the amount of successful attacks per

[2] The source code was available before July 21, 2014, at http://shader.kaist.edu/sslshader. (It has been removed since then, as SSLShader is becoming a commercial product).

**ALGORITHM 2:** The pseudocode of the attacking process in the AES case study

```
Input:
$\vec{M}$: The plaintext
$l$: The length of the plaintext
$\vec{K}$: array of identifiers of the encryption key
Output: TRUE if the attack succeeds, FALSE otherwise
FindLeakage($\vec{M}$,$l$,$\vec{K}$)
begin
    s ← size of current allocable global memory on GPU
    P ← cudamalloc(s)
    $j \leftarrow 0$
    while $j < s$ do
        w ← P[j]
        if $w \in \vec{K}$ then
            cudamemset(P,0,s) /*zeroizing*/
            return TRUE;
        else if $w == M[0]$ then
            $i \leftarrow 0$
            while $i < l$ do
                if $P[j+i]$ != $M[i]$ then
                    cudamemset(P,0,s) /*zeroizing*/
                    return FALSE
                i++
            end
            cudamemset(P,0,s) /*zeroizing*/
            return TRUE
        j++
    end
    cudamemset(P,0,s)
    return FALSE
end
```

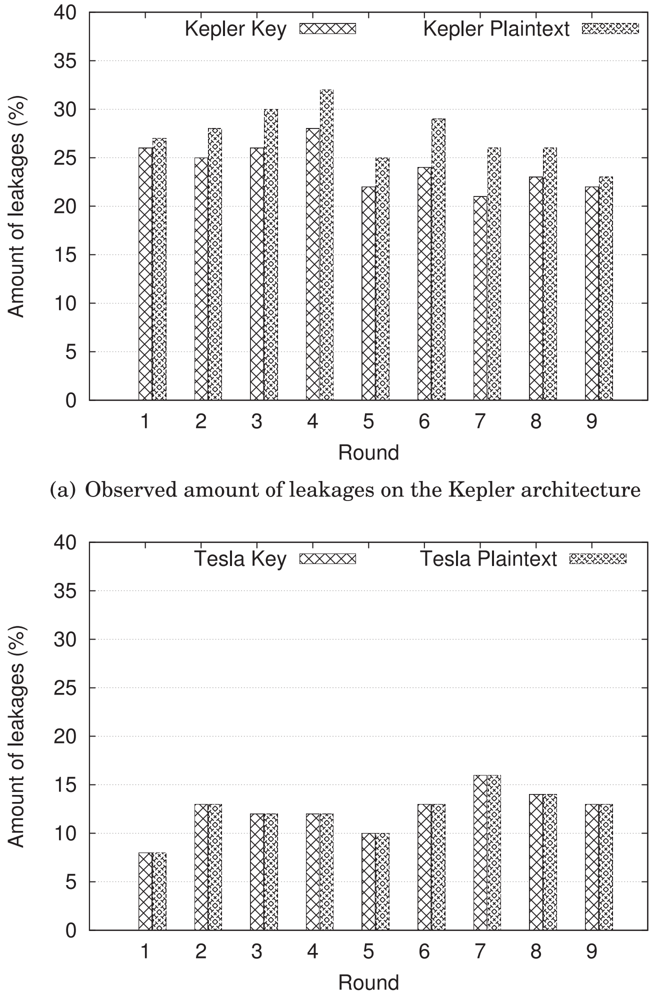
round. To preserve the independence across different rounds, at the end of each round we rebooted the machine.

For the Kepler, we measured a successful attack mean equal to 30% with a standard deviation equal to 0.032. As for the Fermi architecture, we measured a mean success rate of 12% with a standard deviation of 0.03. Figure 9(a) details the results of nine randomly chosen rounds in terms of key leakage and plaintext leakage for Kepler. The plaintext leakage is slightly more frequent than the key leakage. Figure 9(b) shows the results for Fermi; in this case, the frequencies of the two leakages are equal.

Please note that the attacking process we implemented zeroizes [Lee et al. 2005] the memory where the leakage is found. This mechanism was implemented to allow unbiased tests, as it avoids counting the same leakage more than once. In fact, the adversary gets no additional information if the same plaintext is leaked more than once.

### 5.1. Discussion and Qualitative Analysis

It is worth adding further considerations about the conditions that lead to an effective information leakage. Indeed, with our attack methodology, we are able to leak only the final state of the previous GPU process. This limitation is due to the exclusive access granted by the driver to host threads that access the GPU; only one *cudaContext* is allowed to access the GPU at a given time.



(a) Observed amount of leakages on the Kepler architecture

(b) Observed amount of leakages on the Fermi architecture

Fig. 9. Each bin represents the number of times that the leakage occurred over 100 runs of Algorithm 2. We report the results for nine rounds of this experiment.

However, note that in some circumstances, the final state of a computation is sensitive (e.g., the decryption of a ciphertext, the output of a risk-analysis function, or the Universal Transverse Mercator (UTM) locations of an oil well). In other circumstances, the final state of a computation is not sensitive or even public. For instance, knowing the final state of an encryption process (i.e., the ciphertext) does not represent a threat.

However, in our experiments, we were able to recover the original plaintext even after the encryption process ended. This was possible because the plaintext and the ciphertext were stored in different locations of the global memory. As such, this vulnerability depends on both the implementation and the computed function and does not hold in the general case.

Another important consideration about the presented case study concerns the precondition of the attack. To perform the key leakage test, we assume that the adversary knows a portion of the key (which is needed to perform searches in memory and detect if the leakage happened). However, as shown in Riccardi et al. [2013], it is possible to exploit the high entropy of the encryption keys to restrict the possible candidates to a reasonable number. In fact, secure encryption keys usually have a higher entropy than other binary data in memory.

As a further technical note, we found out that SSLShader [Jang et al. 2011a] (i.e., the widespread publicly available implementation of cryptographic algorithms on a GPU) makes use of the *cudaHostAlloc* CUDA primitive. This primitive allocates a memory area in the host memory that is page locked and accessible to the device (pinned memory). The *cudaHostAlloc* can be considered more secure than the *cudaMalloc*, as it uses a pinned page memory that can be accessed by both the device and host. However, we found that even the *cudaHostAlloc* primitive is fully vulnerable to information leakage. In fact, the CUDA runtime copies the data to the GPU global memory on behalf of the programmer when it is more convenient. This is important, as it shows that even code implemented by "experts" actually shows the same deficiencies with regard to security. This finding, together with the others reported in the article, call for solutions to this severe vulnerability.

## 6. PROPOSED COUNTERMEASURES

In previous sections, the main issues and vulnerabilities of Kepler and Fermi CUDA architectures have been highlighted. It is worth noting that discovered leakages are not tied to a particular version of the device driver or GPU architecture. They are intrinsic to present GPU architectures that aim at performance without considering security.

Given that shared-GPU security issues will be increasingly relevant in the future, this section suggests alternative approaches and countermeasure that prevent or at least dramatically limit the described information leakage attacks.

In the following, for each of the discovered vulnerabilities, we provide related mitigation countermeasures.

### 6.1. Shared Memory

As for the shared memory leakage shown in Section 4.2, the proposed fix makes use of a memory-zeroizing mechanism. As already pointed out in Section 4.2, the shared memory attack is ineffective once the host process terminates. The vulnerability window goes from kernel completion to host process completion. As a consequence, the memory-zeroizing operation is better executed inside the kernel. In our opinion, this is a sensitive solution since shared memory is an on-chip area that cannot be directly addressed or copied by the host thread. As such, it is not possible to make use of it from outside a kernel function.

To measure the overhead that an in-kernel memory-zeroizing approach would have on a real GPU, we developed and instrumented a very simple CUDA code (addition of two vectors). Two kernel functions, $K_1$ and $K_2$, were developed: $K_1$ receives as input two randomly initialized vectors $A$, $B$; $K_1$ sums the two vectors and stores the result in vector $C$; $K_2$ is the same as $K_1$ but in addition, it "zeroizes" the shared memory area by overwriting it with the value read from $A[0]$.[3] We measured the execution time difference between $K_1$ and $K_2$ by varying the vectors' size, as this experiment was just aimed at evaluating the scalability of the zeroizing operation. In particular, for $K_1$ and $K_2$, we performed this experiment accessing an increasing number of locations up to the maximum available shared memory. Such value depends on the GPU capability and corresponds to 672KB for the Tesla C2050 and 32KB for the GT640. We noticed that the introduced overhead was constant and not affected by the number of memory accesses.

[3] We did not actually "zeroize" the memory using the value 0 to prevent the compiler from performing optimization, which would have affected the result.

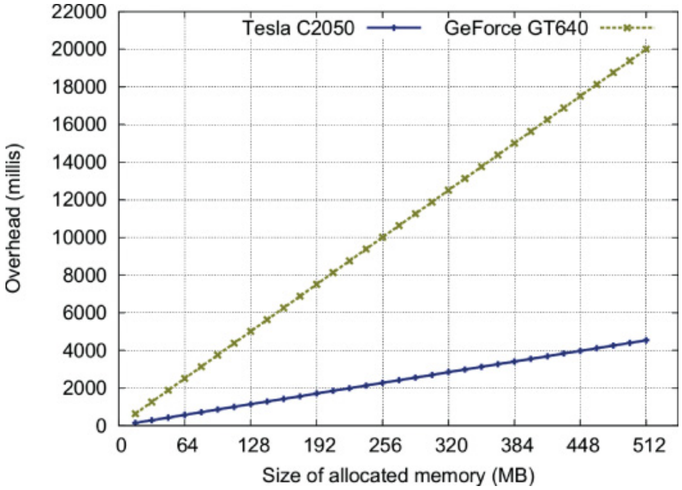


Fig. 10. Overhead introduced by the proposed countermeasure for the global memory leak.

In particular, we measured a mean overhead of 1.66ms on the Kepler and 0.27ms on the Tesla card. We can conclude that for kernels with a reasonable duration (e.g., longer than 0.1 seconds), the proposed fix can be applied without noticeably affecting GPU performance.

### 6.2. Global Memory

As described in Section 4.3, accessing global memory through CUDA primitives can cause an information leakage. The natural fix would consist of zeroizing memory before it is given to the requesting process. This way, when information is deleted, the malicious process is not able to access it. This approach should naturally be implemented inside the CUDA runtime. To assess the impact of the overhead introduced by this solution, we measured the overhead that the same zeroizing operation imposes to traditional memory allocation. The CUDA runtime function *cudaMemset* was used for zeroizing memory content. An incremental size buffer was tested in the experiments. In our tests, size ranged from 16MB to 512MB, in steps of 16MB. We then measured the overhead induced by the additional zeroizing operations. To achieve this goal, we instrumented the source code with the *EventManagement* runtime library function. Through these primitives, we were able to compute the time elapsed between two events in milliseconds with a resolution of around 0.5ms.

As shown in Figure 10, the introduced overhead is not negligible. On both Tesla and Kepler platforms, the induced overhead shows a linear relationship to the allocated buffer size. However, Tesla's line steepness is much lower than the Kepler counterpart, because the two GPUs feature a much different number of multiprocessors (14 for Tesla vs. 2 for the Kepler). In other words, the level of achievable parallelism is quite different.

It is worth noting that a low-level hardware approach would surely be faster. However, in general, zeroizing does worsen performance in the GPU [Yang et al. 2011], as these techniques force additional memory copies between the host and device memory. In addition, we only have implementation details on global memory that is actually implemented on commodity GDDRx memory (i.e., as standard host memory). Introducing an additional mechanism to perform smart memory zeroizing would require an overall redesign of the GDDR approach, and as such it will most probably increase RAM cost. Hardware-based fast zeroizing would probably be the most feasible and convenient solution. However, inner details about low-level memory implementation for CUDA cards are only known by NVIDIA.

Pertaining to the selective deletion of sensitive data, selectively zeroizing specific memory areas is potentially feasible and would potentially reduce unnecessary memory transfers between the GPU and CPU, as most data would not have to be transferred again. A “smart” solution would probably be the addition of CUDA language extensions (source code tags) to mark the variables/memory areas that have to be zeroized since containing sensitive data. On the one hand, this would require language/compiler modifications, whereas on the other hand, it would save some costly data transfers. However, this approach implies some caveats, as there is the risk of pointing the adversary exactly to the memory and registers where sensitive data is located. Further, such sensitive data, when in transit between the CPU and GPU, crosses various memory areas that are still potentially accessible. As such, for performance sake, sensitive areas should be as contiguous as possible.

### 6.3. Registers

Register allocation is handled at the lower level of the software stack, and hence we understand that this leak is due to a flaw regarding the memory isolation implementation. Therefore, fixing this leakage at the application level is quite difficult. A much simpler workaround would be to implement the fix at the GPU driver level. Unfortunately, given the closed-source nature of the driver, at present only NVIDIA can provide a solution for this issue. In particular, the driver should preserve the following properties: first, the registers should not spill to locations in global memory that are still reserved for host threads, and second, the locations of the spilled registers must be reset to zero when they are released.

### 6.4. Discussion

As shown earlier, the overhead introduced by our countermeasures is negligible in most cases. However, it could still be too high in some performance-critical applications. This is why we suggest limiting the adoption of these countermeasures to scenarios where sensitive information is being processed.

In general, from the software point of view, CUDA code writers should pay attention to zeroizing memory as much as possible at the end of kernel execution. Unfortunately, this is troublesome for several reasons:

- Most often, the programmer does not have fine control over kernel code (e.g., if the kernel is the outcome of high-level programming environments such as JavaCL and JCUDA).
- The kernel programmer usually aims at writing the fastest possible code without devoting time to address security/isolation issues that might hamper performance.

As such, we believe that the best results can be obtained if security enhancements are performed at the driver/hardware level. From the CUDA platform/hardware point of view, suggestions include the following:

- Finer GPU hardware–based memory protection mechanisms have to be introduced to prevent concurrent kernels from reading other kernels’ memory.
- Finer monitoring and access control: CUDA should allow the OS to monitor usage and to control access to GPU resources; this way, it could be easier to detect suspicious access patterns and prevent anomalous resource usage.

## 7. RELATED WORK

The rise of multicore architectures across all computing domains has opened the door to heterogeneous multiprocessors. GPUs, in particular, are becoming very popular for speeding up compute-intensive kernels of scientific, imaging, and simulation applications.

For most of the applications that make use of dedicated coprocessors, resources are not highly utilized due to the lack of sustained data parallelism that often occurs in dynamic environments. Beldianu and Ziavras [2013] discussed design frameworks for sharing computation in multicore environments aimed at maximizing core utilization and performance with respect to energy cost. They propose shared work policies that enforce coarse-grain, fine-grain, and vector-lane sharing. [Paolieri et al. 2013] focus on the memory system and analyze delays that a memory request can suffer due to memory interferences generated by co-running tasks. Such interferences might potentially be used to infer memory content, and albeit the issue is not investigated further by Paolieri et al. With regard to GPU memory issues, caches in particular introduce two relevant problems for embedded systems. They consume a significant amount of power, and cache outcomes in multitasking environments are difficult to predict. Reddy and Petrov [2010] proposed a technique that leverages configurable data caches to address the problem of energy inefficiency and intertask interference in multitasking embedded systems. Their approach is aimed at energy efficiency and does not consider information leakage issues.

In fact, information leakage is a serious problem that has been investigated in different scenarios [D'Arco and Perez del Pozo 2013]. In addition, the incidence of malicious code and software vulnerability exploits on embedded platforms is constantly on the rise, as shown by Aaraj et al. [2011], evaluating a malware prevention framework for embedded systems using dynamic binary instrumentation and runtime execution monitoring. Several attacks exploit the existence of side channels in hardware implementations (see Kocher et al. [1999] and Ors et al. [2004]). A preliminary work by Barenghi et al. [2011] investigated side-channel attacks to GPUs using both power consumption and electromagnetic radiations. The proposed approach can be useful for GPU manufacturers to protect data against physical attacks. However, the attacks presented in this article do not require the adversary to have either physical access to the hardware or root privileges.

With regard to secure data deletion, Reardon et al. [2013] present a taxonomy of the characteristics of secure deletion approaches. They suggest that the best approach strongly depends on the used medium. Kindratenko et al. [2009] discuss the use of GPUs in computing clusters. They realize that applications that use GPUs can frequently leave them in an unusable state. To address such issue, prior to node deallocation, a node health check and memory scrubber tool is run that allocates all available GPU device memory and fills it in with a user-supplied pattern. Kindratenko et al. suggest that some security issues for shared-GPU computing exist, but it does not investigate possible information leakage. The suggested coarse-grained workaround uses a wrapper library [Guochun 2012] to perform some basic memory scrubbing at the end of the context.

An interesting work by Maurice et al. [2014] focuses on potential information leakage in virtualized and cloud computing environments. They aim at investigating the causes behind the leakage. However, their paper does not detail the proposed attacks, and it is not clear whether leakages require privileged access. The present article introduces and details a different set of information leakages that do not need root access.

Some of the attack methods discussed in this work bear some similarity with those found in the same time frame and independently from us by Lee et al. [2014], as both studies share the same target GPU architecture (CUDA). Nevertheless, our work, as detailed in Sections 3 and 4, provides several novel contributions over Lee et al. [2014]. In particular, we consider and study multiple shared memory scenarios. In fact, we investigate and describe the behavior of the scheduler with respect to the multiprocessor and shared memory in Section 4.2. Further, Lee et al. do not consider attacks based on GPU registers, which are particularly relevant for their impact on global memory data, as discussed in Section 4.4. Finally, we also propose, discuss, and evaluate possible countermeasures to the discovered security issues and consider the actual SSLShader [Jang et al. 2011a] case study.

Most GPU manufacturers are reluctant to publish the internals of their solutions. In fact, documentation is mostly generic, marketing oriented, and incomplete. This fact hinders the analysis of the information leakage problem on GPUs. As a consequence, in the literature, most of the available architectural information over existing hardware is due to black-box analysis. In particular, Wong et al. [2010] developed a microbenchmark suite to measure architectural characteristics of CUDA GPUs. The analysis showed various undisclosed characteristics of the processing elements and the memory hierarchies and exposed undocumented features that impact both program performance and program correctness. CUBAR [Black and Rodzik 2010] used a similar approach to discover some of the undisclosed CUDA details. In particular, CUBAR showed that CUDA features a Harvard architecture on a Von Neumann unified memory. Further, since the closed-source driver leverages (the deprecated) security through an obscurity paradigm, inferring information from a PCIe bus [NVIDIA 2014a] is possible, as partially shown in Kato [2012].

GPU thread synchronization issues are introduced and discussed in Feng and Xiao [2010], whereas Shye et al. [2009] describe multicore computing reliability issues. The two preceding works are aimed toward correctness, reliability, and performance, whereas in our work we focus on actual GPU thread behavior and related consequences on data access.

Vulnerabilities have been discovered in the past in the NVIDIA GPU driver [Larabel 2012], a key component of the CUDA system that has OS kernel[4] level access (via the NVIDIA OS kernel module). Such vulnerabilities can have nasty effects on the whole system and can lead to even further information leakage, due to root access capabilities. Recently, the digital forensic community focused on the topic of GPU-assisted malware. Villani et al. [2015] discuss the impact on memory forensics of malware that leverages the GPU for antiforensic purposes. In particular, the authors present an interesting case study on Intel-integrated GPUs.

With respect to Web rendering security issues, Kotcher et al. [2013] discussed the possible leakage related to using CUDA programs to access graphical content rendered by the GPU. However, Kotcher et al. have a totally different target than our own, as we aim to access computations and data related to GPGPU computing.

Finally, a relevant limitation of current GPU architectures is the fact that the OS is completely excluded from the management of GPU threads. The first attempts to overcome the limits of present GPU platforms aim to give the OS kernel the ability to control the marshalling of GPU tasks [Kato et al. 2011]. In fact, the GPU can be seen as an independent computing system where the OS role is played by the GPU device driver; as a consequence, host-based memory protection mechanism are actually ineffective to protect GPU memory.

## 8. CONCLUSION AND FUTURE WORK

In this work, we provide several contributions, shedding light on some security issues of the increasingly successful (GP)GPU computing field. In particular, we detail some

[4] We refer here to the OS kernel to avoid ambiguity with CUDA kernels.

critical vulnerabilities in CUDA architectures affecting shared memory, global memory, and registers. We also experimentally show how such vulnerabilities can be exploited to generate information leakage. Furthermore, fixes are proposed and discussed to tackle the highlighted vulnerabilities.

Given the generality of the identified vulnerabilities and the architectural complexity of the GPU field, the results reported in this article—other than being interesting on their own—also pave the way for further research.

## REFERENCES

Najwa Aaraj, Anand Raghunathan, and Niraj K. Jha. 2011. A framework for defending embedded systems against software attacks. *ACM Transactions on Embedded Computing Systems* 10, 3, Article No. 33.

Alessandro Barenghi, Gerardo Pelosi, and Yannick Teglia. 2011. Information leakage discovery techniques to enhance secure chip design. In *Information Security Theory and Practice: Security and Privacy of Mobile Devices in Wireless Communication*. Lecture Notes in Computer Science, Vol. 6633. Springer, 128–143.

Spiridon F. Beldianu and Sotirios G. Ziavras. 2013. Multicore-based vector coprocessor sharing for performance and energy gains. *ACM Transactions on Embedded Computing Systems* 13, 2, Article No. 17.

Nick Black and Jason Rodzik. 2010. My Other Computer Is Your GPU: System-Centric CUDA Threat Modeling with CUBAR. Retrieved December 26, 2015, from http://nick-black.com/dankwiki/images/d/d2/Cubar2010.pdf.

Lilian Bossuet, Michael Grand, Lubos Gaspar, Viktor Fischer, and Guy Gogniat. 2013. Architectures of flexible symmetric key crypto engines—a survey: From hardware coprocessor to multi-crypto-processor system on chip. *ACM Computing Surveys* 45, 4, Article No. 41.

Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. 2004. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics* 23, 3, 777–786.

Wu Chun Feng and Shucai Xiao. 2010. To GPU synchronize or not GPU synchronize? In *Proceedings of the 2010 IEEE International Symposium on Circuits and Systems (ISCAS'10)*. 3801–3804.

A. Di Biagio, A. Barenghi, G. Agosta, and G. Pelosi. 2009. Design of a parallel AES for graphics hardware using the CUDA framework. In *Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS'09)*. 1–8.

Jack Dongarra, Erich Strohmaier, and Horst Simon. 1993. TOP500 Supercomputing Sites. Retrieved December 26, 2015, from http://www.top500.org.

Paolo D'Arco and Angel Perez del Pozo. 2013. Toward tracing and revoking schemes secure against collusion and any form of secret information leakage. *International Journal of Information Security* 12, 1, 1–17.

Donald Evans, Phillip Bond, and Arden Bement. 1994. FIPS PUB 140-2: Security Requirements for Cryptographic Modules. Available at http://www.csrc.nist.gov.

Abhijeet Gaikwad and Ioane Muni Toke. 2010. Parallel iterative linear solvers on GPU: A financial engineering case. In *Proceedings of the 18th Euromicro PDP Conference.* IEEE, Los Alamitos, CA, 607–614.

Shi Guochun. 2012. CUDA Wrapper Library. Available at http://cudawrapper.sourceforge.net.

Michael Henson and Stephen Taylor. 2014. Memory encryption: A survey of existing techniques. *ACM Computing Surveys* 46, 4, Article No. 53.

Howard M. Heys. 2002. A tutorial on linear and differential cryptanalysis. *Cryptologia* 26, 3, 189–221.

Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. 2011b. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel and Distributed Systems* 22, 1, 105–118.

Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and Kyoung Soo Park. 2011a. SSLShader: Cheap SSL acceleration with commodity processors. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. 1.

Shinpei Kato. 2012. Gdev. Retrieved December 26, 2015, from https://github.com/shinpei0208/gdev.

Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa. 2011. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX Annual Technical Conference (USENIXATC'11)*. 2–16.

Junsung Kim, Ragunathan (Raj) Rajkumar, and Shinpei Kato. 2013. Towards adaptive GPU resource management for embedded real-time systems. *ACM SIGBED Review* 10, 1, 14–17.

V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W.-M. Hwu. 2009. GPU clusters for high-performance computing. In *Proceedings of the IEEE International Conference on Cluster Computing and Workshops (CLUSTER'09)*. 1–8. DOI:http://dx.doi.org/10.1109/CLUSTR.2009.5289128

Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. *Differential Power Analysis*. Springer-Verlag.

Robert Kotcher, Yutong Pei, Pranjal Jumde, and Collin Jackson. 2013. Cross-origin pixel stealing: Timing attacks using CSS filters. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*. ACM, New York, NY, 1055–1062.

Michael Larabel. 2012. NVIDIA Linux Driver Hack Gives You Root Access. Retrieved December 26, 2015, from http://www.phoronix.com/scan.php?page=news_item&px=MTE1MTk.

Ruby B. Lee, Peter C. S. Kwan, John P. McGregor, Jeffrey Dwoskin, and Zhenghong Wang. 2005. Architecture for protecting critical secrets in microprocessors. *SIGARCH Computer Architecture News* 33, 2, 2–13.

Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. 2014. Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P'14)*.

Flavio Lombardi and Roberto Di Pietro. 2010. CUDACS: Securing the cloud with CUDA-enabled secure virtualization. In *Proceedings of the 12th International Conference on Information and Communications Security (ICICS'10)*. 92–106.

Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2014. Confidentiality issues on a GPU in a virtualized environment. In *Proceedings of the 18th International Conference on Financial Cryptography and Data Security (FC'14)*.

Rebecca T. Mercuri and Peter G. Neumann. 2003. Security by obscurity. *Communications of the ACM* 46, 11, 160–166.

Paulius Micikevicius. 2011. Local Memory and Register Spilling. Retrieved December 26, 2015, from http://on-demand.gputechconf.com/gtc-express/2011/presentations/registe r_spilling.pdf.

N. Nishikawa, K. Iwai, and T. Kurokawa. 2011. High-performance symmetric block ciphers on CUDA. In *Proceedings of the 2011 2nd International Conference on Networking and Computing (ICNC'11)*. 221–227.

NVIDIA. 2014a. CUDA C Programming Guide. Retrieved December 26, 2015, from http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

NVIDIA. 2014b. GRID GPUs. Available at http://www.nvidia.com/object/grid-technology.html.

S. B. Ors, F. Gurkaynak, E. Oswald, and B. Preneel. 2004. Power-analysis attack on an ASIC AES implementation. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04)*, Vol. 2. 546–552. DOI:http://dx.doi.org/10.1109/ITCC.2004.1286711

Marco Paolieri, Eduardo Quinones, and Francisco J. Cazorla. 2013. Timing effects of DDR memory systems in hard real-time multicore architectures: Issues and solutions. *ACM Transactions on Embedded Computing Systems* 12, 1, Article No. 64.

Alexandros Papakonstantinou, Karthik Gururaj, John A. Stratton, Deming Chen, Jason Cong, and Wen-Mei W. Hwu. 2013. Efficient compilation of CUDA kernels for high-performance computing on FPGAs. *ACM Transactions on Embedded Computing Systems* 13, 2, Article No. 25.

Joel Reardon, David Basin, and Srdjan Capkun. 2013. SoK: Secure data deletion. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'13)*. IEEE, Los Alamitos, CA, 301–315.

Rakesh Reddy and Peter Petrov. 2010. Cache partitioning for energy-efficient and interference-free embedded multitasking. *ACM Transactions on Embedded Computing Systems* 9, 3, Article No. 16.

Marco Riccardi, Roberto Di Pietro, Marta Palanques, and Jorge Aguilí Vila. 2013. Titans' revenge: Detecting Zeus via its own flaws. *Computer Networks* 57, 2, 422–435.

Alex Shye, Joseph Blomstedt, Tipp Moseley, Vijay Janapa Reddi, and Daniel A. Connors. 2009. PLR: A software approach to transient fault tolerance for multicore architectures. *IEEE Transactions on Dependable and Secure Computing* 6, 2, 135–148.

Henk C. A. Van Tilborg and Sushil Jajodia (Eds.). 2011. *Encyclopedia of Cryptography and Security* (2nd ed.). Springer.

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)*. ACM, New York, NY, 1131–1142.

Uri Verner, Assaf Schuster, and Mark Silberstein. 2011. Processing data streams with hard real-time constraints on heterogeneous systems. In *Proceedings of the International Conference on Supercomputing (ICS'11)*. ACM, New York, NY, 120–129.

Antonio Villani, Davide Balzarotti, and Roberto Di Pietro. 2015. The impact of GPU-assisted malware on memory forensics: A case study. In *Proceedings of the Annual Digital Forensics Research Conference (DFRWS'15)*.

H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'10)*. 235–246.

Xi Yang, Stephen M. Blackburn, Daniel Frampton, Jennifer B. Sartor, and Kathryn S. McKinley. 2011. Why nothing matters: The impact of zeroing. *ACM SIGPLAN Notices* 46, 10, 307–324.

Zillians. 2012. VGPU GPU virtualization. Available at http://www.zillians.com.

Received September 2014; revised March 2015; accepted July 2015