Analyzing and Improving GPU Security

Sparsh Mittal

IIT Hyderabad, India

Acronyms

- GlM/LoM/ShM = global/local/shared memory
- DoS = denial of service
- ASLR = Address space layout randomization

Covert and side-channel

- Channel= medium through which sensitive data is leaked

- If channel is hidden, it is termed "**covert channel**".
- A covert channel is created intentionally and is not otherwise meant for communication. The adversary tries to conceal its existence from the victim

Covert and side-channel

- A "**side-channel**" is created incidentally, where spy gets sensitive info from characteristics of system's operation
- E.g., if the timing/power values of the encryption algorithm depend on the key, then, based on timing/power measurements, a spy can guess the key
- In a side-channel, there is no communication, but only leakage of sensitive information through the side-channel

Denial-of-service attack

- In a DoS attack, the adversary tries to make the device unavailable by temporarily or permanently hampering its services
- This may be done by overloading the system with useless requests which prohibits handling of genuine requests from a benign user
- Due to this, a DoS attack can be detected
- This is different from other attacks, such as side-channel attack, where the system is generally not harmed and hence, no evidence of the attack remains

Buffer overflow

- Buffer overflow refers to writing data outside the boundary of the buffer to the adjacent locations
- It can lead to program crashes, data corruption, and security breaches
- For example, stack overflow by a thread can impact execution of other threads by overwriting other memory spaces
- **ASLR:** a memory-protection process for OSes that guards against buffer-overflow attacks by randomizing the location where system executables are loaded into memory.

Soft/hard booting and active/passive attacks

- **Soft (or warm) rebooting:** the system is restarted without turning off the power supply to the system
- **Hard (or cold) rebooting:** the power to the system is first turned-off (i.e., shut-down) and then turned-on which results in rebooting the system

- **Passive attacks:** leak system-information but do not the change the system
- **Active attacks:** change the data or operation of the system

Motivation of Improving GPU Security (1 of 2)

- GPUs were originally used for graphics applications
- Now, they are used for mission-critical applications
  - Defense, encryption, financial
- These domains require high security guarantee

Possible attack scenarios

- In companies, a malicious insider may access classified documents which were opened on a shared GPU by an authorized user

- An adversary can rent a GPU-based VM and leak information of users using other VMs on the same system via GPU memory

Researchers have demonstrated …

- Stealing the credit card numbers and email contents from remnant data in GPU memory
- Guessing the opened tabs, address bar and page-body from Google chrome
- Figure-portions and text-lines from recently-opened Adobe Reader documents
- Whole or portions of images from MATLAB.
- Malware which logs keyboard activity for stealing sensitive data

Motivation of Improving GPU Security (2 of 2)

- **ESEA video-gaming company incident [ESEA]**
  - Malicious employee hid bitcoin miner in software
  - This miner used GPUs in users' machines for mining bitcoin without their knowledge.
  - The miner overheated and harmed the machines by overloading the GPUs.
  - Thus, the malicious person earned cryptocurrency at the expense of the users' resources.
- This incident strongly highlights the need of improving GPU security

[ESEA] https://goo.gl/sLDMw8,

Attacking GPUs is challenging

- Many attacks exploit the correlation between an event and its impact such as the change in latency, power consumption or number of memory accesses
- However, due to GPU's massively-parallel architecture and undocumented policies, isolating individual events and their impact is generally not feasible
- Hence, isolating side-channels on GPUs is not easy

Attacking GPUs is challenging

- Only one cudaContext can run on GPU at any time, a data-leakage attack can obtain only the final snapshot of the previous process

- In a cloud environment, both the cloud and GPU architectures offer layers of obscurity which makes it difficult to launch an attack on GPUs

Securing GPUs is also challenging!

1. Limitations of CPU-based security solutions

- After launching the program on GPU, CPU remains isolated. It does not monitor the GPU

- => Security mechanisms proposed on CPUs may not work for GPUs

- e.g., by exploiting this, an attacker can use GPU as the polymorphic malware extractor
  - whereby the host can load the compressed/encrypted code on GPU and then call a GPU kernel to quickly unpack/decrypt the code

1. Limitations of CPU-based security solutions

- A sharp increase in GPU load is likely to go undetected more easily compared to that in CPU load,
- => a GPU malware is more stealthy

- GPU can crack passwords by using a brute force attack

2. Lack of documentation and open-source tools

- GPU vendors take “security-through-obscurity approach”. They do not reveal info about GPU architecture
- Most info obtained through reverse engineering only
- Lack of official docs allows GPU vendors to introduce architectural changes for boosting perf., even at the cost of security
- Docs discuss only perf. and not security-related issues

2. Lack of documentation and open-source tools

- GPU binary utilizes closed-source assembly language which cannot be inspected by existing anti-virus tools
- GPU drivers may not be as rigorously evaluated from the security perspective as the existing OSes
- GPU vendors do not define/document what happens to the deallocated memory
- GPU arch evolving fast
- Reasoning about GPU's security guarantees is difficult

3. Lack of data erasure

- GPU hardware/drivers do not erase their memories and thus, in ShM, LoM, GlM and registers, data persists after deallocation
- By exploiting this, an adversary can leak sensitive info
  - e.g., to leak information from registers, an adversary can write a kernel with the same occupancy and thread-block size as the victim kernel
  - This ensures similar, predictable partitioning of register file
  - Then, the malicious kernel can be coded in a way to read the target registers

4. Increasing reliance on GPUs

- GPUs are increasingly being used for accelerating a wide variety of apps
  - e.g., WebGL allows browsers to utilize GPUs for accelerating webpage rendering

- This can be leveraged to launch DoS attack remotely by making a user open a malicious website which overloads users' GPUs

- Adversary can also change the contents displayed on user's screen

5. Characteristics of GPU architecture

- In GPUs, the presence of multiple memories with different access rights and lifetimes complicates their security solutions and mandates individual security solutions for them

- Even non-privileged users can run GPU programs, a large number of users can attack or exploit GPU

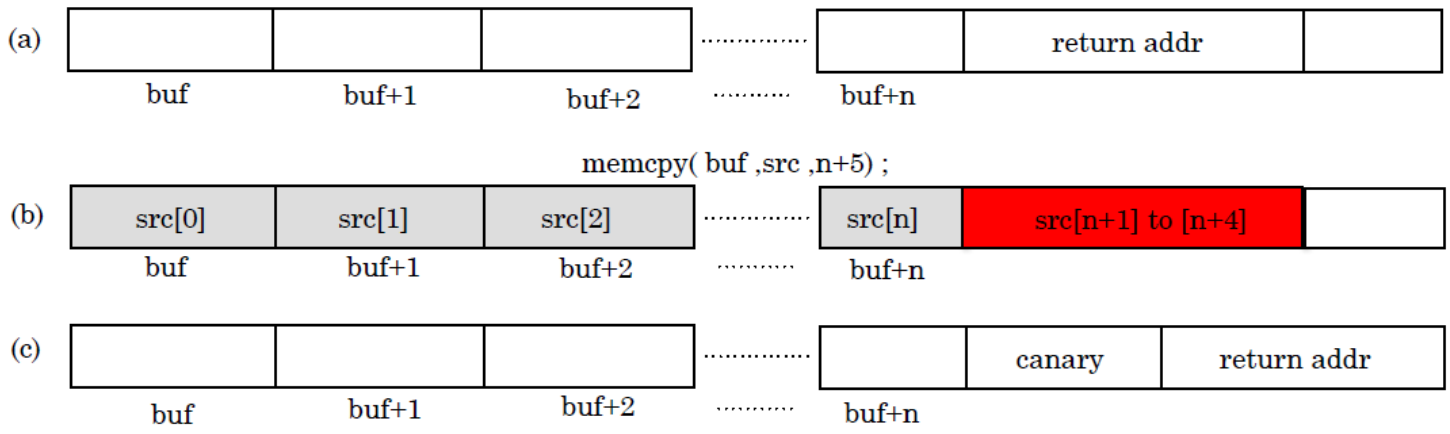
Mitigating Buffer Overflow Attack

Use of canary (1 of 2)

- A canary is the memory location which does not store useful data, and is generally placed right after a buffer which has high likelihood of being overwritten

- In case of buffer overflow, the canary is overwritten first and thus, useful data are saved from being overwritten.

Use of canary (2 of 2)



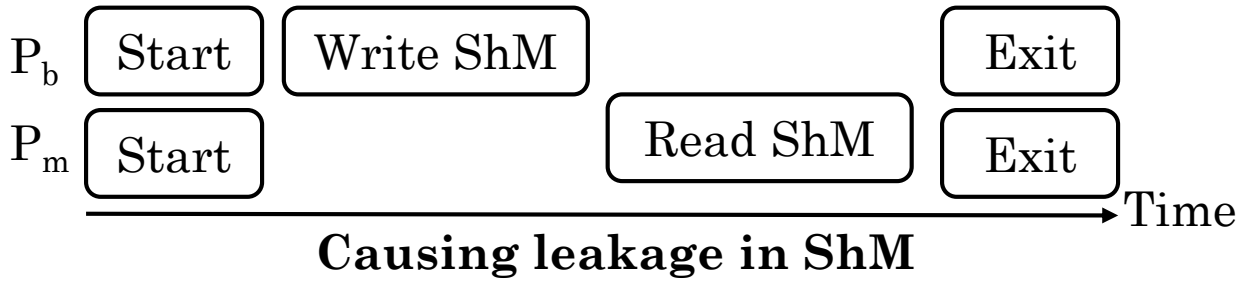
(a) A buffer [0:n]. After the buffer, return address of a function is stored.
(b) An adversary can copy an excessive amount of data to cause overflow and thus, overwrite nearby variables (return address)
(c) Use of a canary value after the buffer. Canary changed => buffer overflow occurred

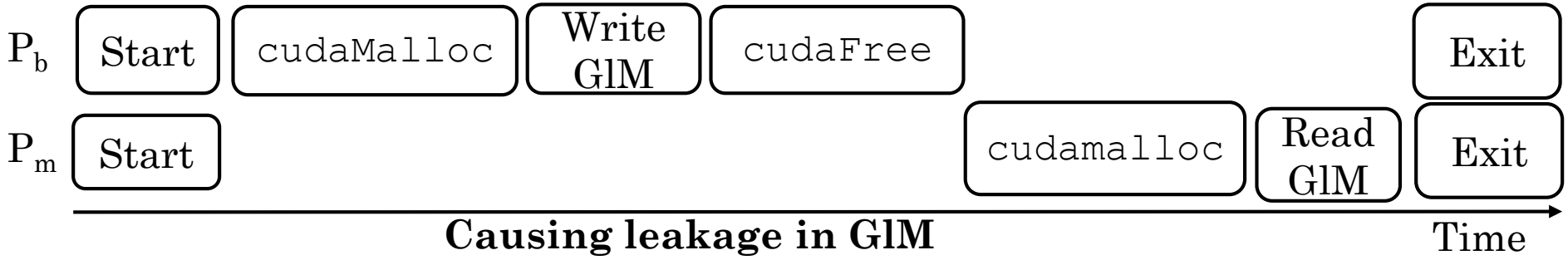
Limitations of use of canary

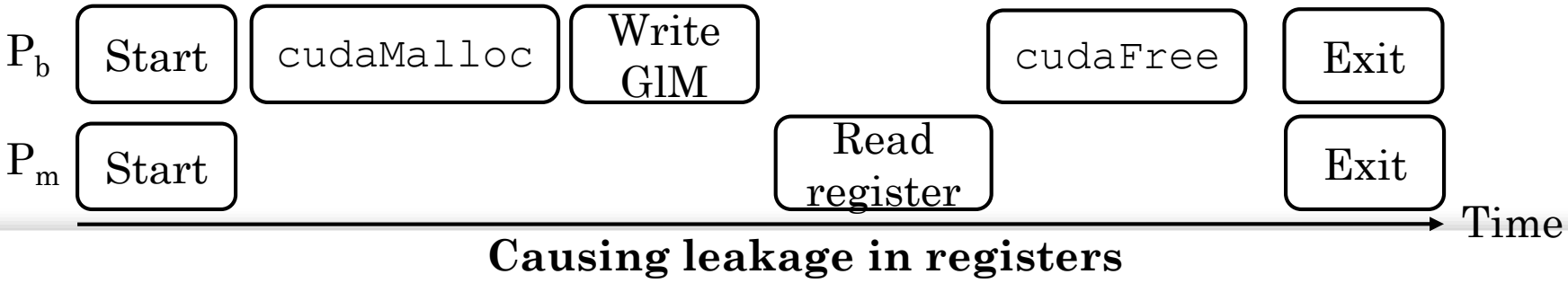
- Canaries are checked only after completion of the kernel
- Before the canaries are checked, a garbage value may get consumed, and the canaries can be reset which avoids detection of overflow.
- An adversary who knows the canary size can overwrite beyond it. Thus, canary do not fully guarantee security
- Use of canaries wastes memory capacity

Leaking data from GPU Memories

Event sequence for causing leakage







Leaking data from registers

- Exploit “register spilling” mechanism
  - **Register spilling:** A process can reserve more registers than those available on GPU. Variables which cannot be placed in the register are placed in GlM. This is termed as register spilling.
- Using this, an attacker can access GlM reserved for other CUDA contexts, even when the benign process owns them and has not freed them using cudaFree
- This makes the attack very dangerous
- Limitations: This attack does not allow interfering with the computations performed by the benign process. This attack was successful on Kepler but not Fermi

Leakage due to no erasure and no virtual memory

- Unlike CPUs, GPUs do not implement ASLR or virtual memory
- On GPUs, pointer allocations repeatedly return the same addresses => The data-address can be determined
- In absence of virtual memory, logical addresses of different processes access the same physical memory
- Since GPUs do not erase the memory, the data persists in memory even after program termination
- This attack succeeded even across different users and login sessions

Leakage through binaries (1 of 2)

- CUDA binaries have high-level assembly PTX code
- This info may be used for malicious purposes or reverse engineering
- By modifying and compiling PTX code in a just-in-time manner, even simple dynamic analysis can reveal details about the source-code
- This, along with debugging tools can extract lot of info

Leakage through binaries (1 of 2)

- **Thwarting this attack**:
- Distribute *GPU-architecture specific binaries* without PTX portion, instead of distributing *platform-independent binaries*
  - Former makes it difficult to perform reverse engineering

- Obfuscate function/variable names

Side-channel attack

We discuss timing side-channel. Power side-channel can be similarly formed

Leaking AES key on GPU

- AES128: 128b AES encryption with T-tables on GPU. It uses 16B key to encrypt a 16B block
- It has 10 rounds of operations. In each round, a 16B key is used. From any round key, one can obtain the original 16B key. So, attacker focuses on leaking the last-round key

- Assume we run AES128 on GPU
- Depending on where the table is stored, we can exploit a side-channel

Storing table in shared or global memory

Table stored in	Shared memory	Global memory
Correlation	Execution time depends on shared memory conflicts, which depends on the address accessed by each thread	Execution time depends on the number of unique cache line requests after coalescing, which depends on the address accessed by each thread

Side-channel attack thru GlM (1 of 3)

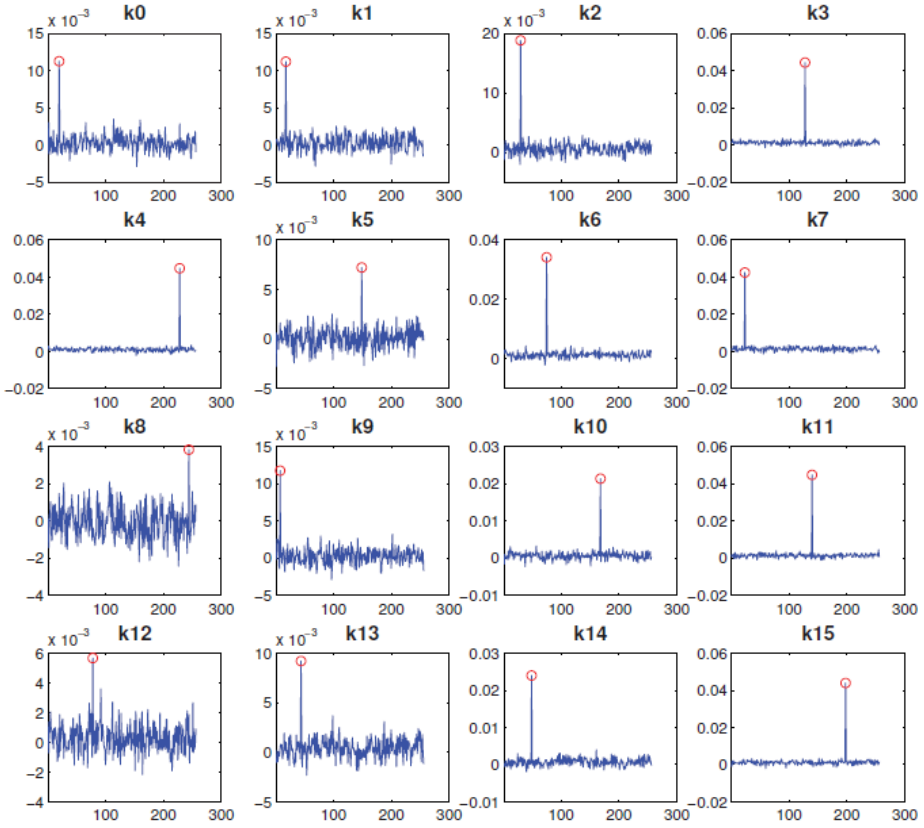
- AES algorithm issues memory requests for loading its T-table entries, whose addresses depend on the plaintext and key
- In GPU, a load instruction of a warp generates one memory request from each of its 32 threads
- These requests are coalesced and merged with the requests queued in miss status holding register (MSHR)
- The time incurred in serving these 32 memory address requests of a warp scales linearly with the number of unique cache line requests

Side-channel attack thru GlM (2 of 3)

- Given this high correlation, memory addresses (and hence, the key) can be inferred from the timing measurements
- Specifically, in the last round of AES, each table index can be obtained from one byte of the key and corresponding byte of ciphertext, irrespective of other bytes of ciphertext
- Using this, each byte can be individually guessed
- For each key guess, compute number of coalesced accesses for every 32-block message, and then, find the correlation of timing with the NCAs

Side-channel attack thru GlM (3 of 3)

- For a correct key-byte guess, the NCA is correct and hence, the correlation is highest, otherwise it is low. From this, the correct key can be estimated



Jiang et al. HPCA'16

Mitigating side-channel attack thru GlM

- More noise/randomness can be added to timing measurements
- Key can be frequently changed
- Mapping of table-lookup index to cache line can be randomized, which prohibits the attacker from deducing the number of unique cache line requests generated

Covert-channel attack

- If GPUs allow multiple kernels to co-reside, they become vulnerable to covert-channels
- Assume two kernels: trojan kernel and spy kernel
- They belong to two different applications and mutually communicate covertly

- Attack procedure: we first ensure that these kernels co-reside and share resources.
- Covert channel can be created using cache, functional unit and GlM

Covert channel using cache

- Cause contention on L1 constant cache, since it is small in capacity
- Run trojan and spy kernels using two streams on GPU
- Trojan encodes ‘1’ or ‘0’ by causing contention or staying idle, respectively
- To cause contention, trojan accesses a single set
- Spy also accesses the same set and records the latency
- **High latency** => value ‘1’ was transmitted since the data was replaced by a trojan
- **Low latency** => value 0 was transmitted

Thwarting Covert Channels

- Partition GPU resources so that communicating kernels cannot measure mutual contention, e.g.,
  - Perform cache partitioning
  - Prohibit co-location of different kernels
- Allow pre-emption of programs
- Introduce randomness in scheduling policies and noise in latency-measurements

- These approaches incur performance overhead and increase complexity

Malware

Malware can escape detection

- Malware can escape detection using 4 strategies
  - Unlimited code-execution
  - Process-less execution
  - Context-less execution
  - Inconsistent memory mapping

Unlimited code execution

- GPUs have non-preemptive execution model => a single task can occupy the GPU completely
- To avoid this, GPU driver uses a timeout scheme (e.g., "hangcheck" function), by which a longlasting process can be killed
- Malware can disable this to occupy the GPU indefinitely

Process-less code execution

- A GPU kernel is usually always controlled by a host process
- The malware may run a kernel without any controlling process on the host by killing the host process right after the GPU kernel starts
- **Limitation:** It still leaves traces in the GPU driver, e.g., the buffer objects and hardware context of the GPU kernel may still be present in the driver's memory

Context-less code execution

- Here, the hardware context of GPU kernel is removed completely from the records of the driver
- Thus, the kernel can hide its presence completely.

- Both process-less and context-less executions require that the kernel has super-user privileges and has already achieved “unlimited code execution”.
- However, the latter additionally requires knowledge about driver internals

Inconsistent memory mapping

- Generally, the list of accessible physical pages is kept both in the OS and GPU memories
- However, the information kept in these two-page tables can be made to differ
- Then, the OS page table points to the correct page, but the GPU address points to a random memory location

Denial-of-service attack

Launching DoS attack

- Assign a long-running task on GPU which makes it unavailable for other system-tasks
- Since a GPU task cannot be pre-empted, the system becomes unresponsive
- DoS attacks can be launched using graphics APIs, eg., DirectX, OpenGL and WebGL

- To attack vertex and fragment shader, an infinite-loop can be added in them

Attacking drawing (gl.draw) function

- Instead of calling it multiple times, it should be called only once and given a large amount of workload, e.g., many complex shapes
- Reason: after each function invocation, control returns to CPU which would foil the DoS attack

- Number of shapes should be
  - small enough to fit in the GPU memory and avoid a GPU-crash and
  - large enough to make GPU unresponsive while rendering them

Results of DoS attacks on different GPUs and operating systems

	Nvidia GPU	ATI GPU	Intel GPU
Results of flooding the `gl.draw` function			
Windows XP	Total system freeze	System freeze, then GPU recovery message	Not tested
Windows 7	System freeze, then graphics driver reset	System freeze, then graphics driver reset. Occasional total system freeze.	System freeze, then graphics driver reset.
Mac OS X	Total system freeze	Total system freeze	Not tested
Red Hat Linux	System freeze, then graphics driver reset		
Results of the attack on vertex/fragment shader			
Windows XP	Total system freeze	System freeze, then GPU recovery message	Not tested
Windows 7	System freeze, then graphics driver reset		
Mac OS X	Total system freeze	Total system freeze	Not tested
Red Hat Linux	Vertex shader: total system freeze. Fragment shader: hang, then graphics driver reset		

Thwarting DoS attacks

- Some OSes use timers to ascertain when GPU stops responding and then reset the driver to reclaim GPU
  - This mechanism should be made perfect; currently it is not
- By performing static analysis, the runtime can be estimated and if it exceeds a threshold, the kernel launch can be prohibited
- By allowing simultaneous execution of multiple tasks on GPUs, some resources can be ensured for OS processes, even if other task stops responding
- Resetting should be performed in time to avoid complete system crash