GPU Memory Exploitation for Fun and Profit

USENIX Security 2024

Yanan Guo*, Zhenkai Zhang*, Jun Yang

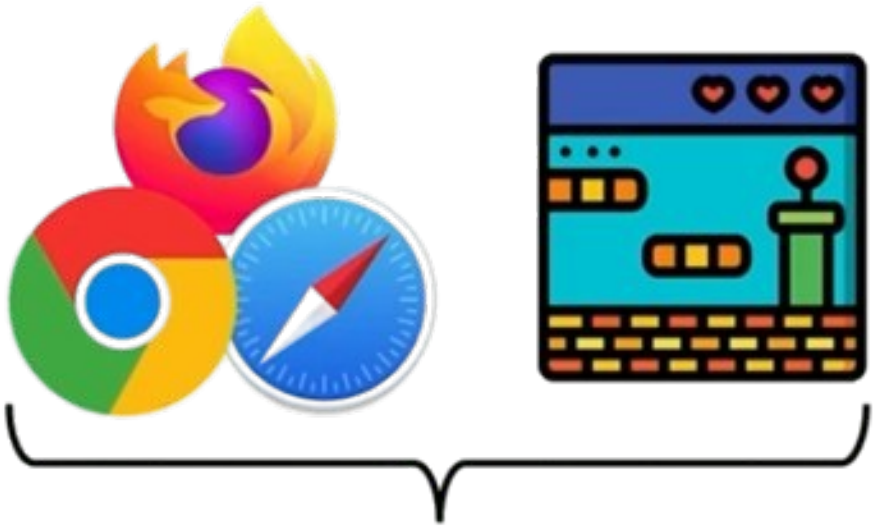
University of Rochester, Clemson University, University of Pittsburgh

Presented by: Marios Mongogiannis

Table of Contents

Introduction

We all know that GPU stands for “Graphics” Processing unit



Graphics Processing

APIs: OpenGL, WebGL, DirectX…

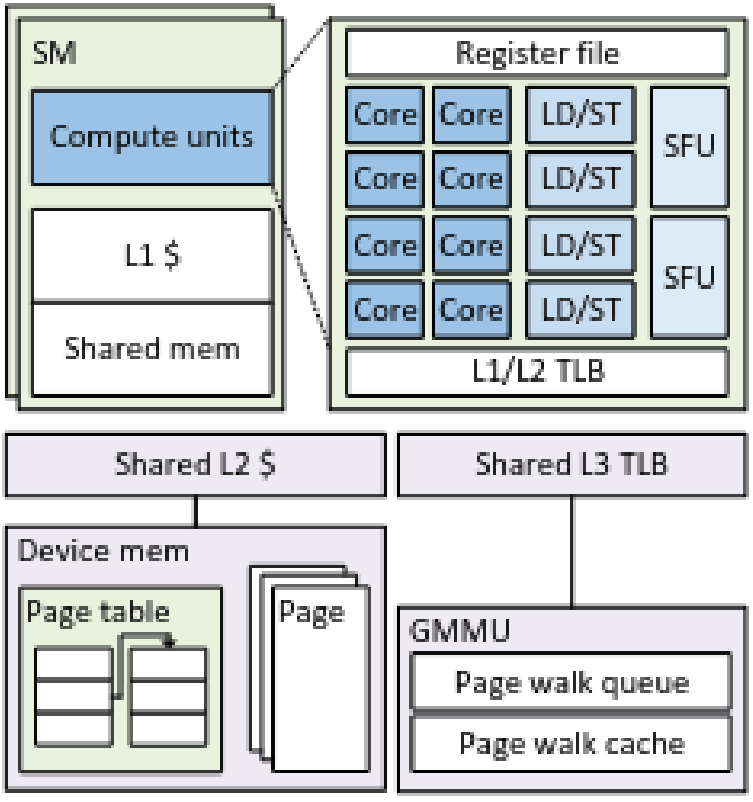
General-Purpose Computing

APIs: CUDA, OpenCL…

Introduction

What about Security?

GPU Basics & Memory Architecture



GPU Basics & Memory Architecture

- Global Memory
- Local Memory (thread private)
- Shared Memory
- Texture Memory
- Constant Memory

```
/** device function **/
__device__ void add(char* d_global){
    d_global[0] += 1;
}

/** CUDA kernel **/
__global__ void mem_type(char* d_global){
    char d_local[10];
    d_local[0] = d_global[0];
    __shared__ d_shared[10];
    d_shared[0] = d_global[0];
    add(d_global);
}

/** CPU function, calling the cuda kernel **/
void kernel_launch (char* d_cpu){
    char* d_global;
    /** Allocate a GPU buffer **/
    cudaMalloc(&d_global, 1024);
    cudaMemCpy(d_cpu, d_global, 1024,
                cudaMemcpyHostToDevice);
    mem_type <<<8,32>>> (d_global);
}
```

GPU Basics & Memory Architecture

Pointer	Memory type	Storage	Cached	Load/store instructions	Scope
d_global	Global memory	Device memory (off chip)	Yes	LDG/STG	Process
d_local	Local memory	Device memory (off chip)	Yes	LDL/STL	Thread
d_shared	Shared memory	Shared memory (on chip)	No	LDS/STS	Thread block

+Generic LD/ST instructions which can be used for accessing all memory spaces

GPU Basics & Memory Architecture

Prior studies have revealed that that memory errors such as buffer overflows can occur within a specific memory space like an OOB operation on a local memory buffer can compromise data in the same local memory. However they haven’t explored whether such operation can occur across memory spaces.

Additionally they concluded that using buffer overflow attacks to hijack the control flow is difficult because the return address is stored in an undisclosed memory location, not on the stack.

They also said that traditional code injection attacks cannot be applied against CUDA programs because code and data are separated in memory.

Demystifying GPU Memory & Out-of-Bounds Operations

Their Conclusions do not hold!!!

Demystifying GPU Memory & Out-of-Bounds Operations

Global Memory

- LDG/STG with 49-bit addr

```
/** R6.64:0x7fffcda00000 **/
LDG R8,[R6.64]
STG [R6.64],R8
```

Local Memory

- LDL/STL with 24-bit addr

```
/** R6:0xfffd80 **/
LDL R8,[R6.64]
STL [R6.64],R8
```

Generic Pointer

- LD/ST with 49-bit addr
- Local mem: prefix||local_addr

  0xfffd80 ⟶ 0x**7ffff2**fffd80

Demystifying GPU Memory & Out-of-Bounds Operations

- LDG/STG with a local mem addr

```
/** R6.64:0x7ffff2fffd80 **/
LDG R8,[R6.64]
STG [R6.64],R8
```

Error: illegal memory access

Demystifying GPU Memory & Out-of-Bounds Operations

```
__global__ kernel_example{
    int arr[10] = {thread_id};
    printf("addr %p, data %d\n", arr, arr);
}
```

Launch the kernel with 32 threads

```
addr 0x7ffff2fffd80, data 0
addr 0x7ffff2fffd80, data 1
addr 0x7ffff2fffd80, data 2
addr 0x7ffff2fffd80, data 3
...
```

HOW

- Same virtual addr, different data!
- Guarantees thread-private local mem

Demystifying GPU Memory & Out-of-Bounds Operations

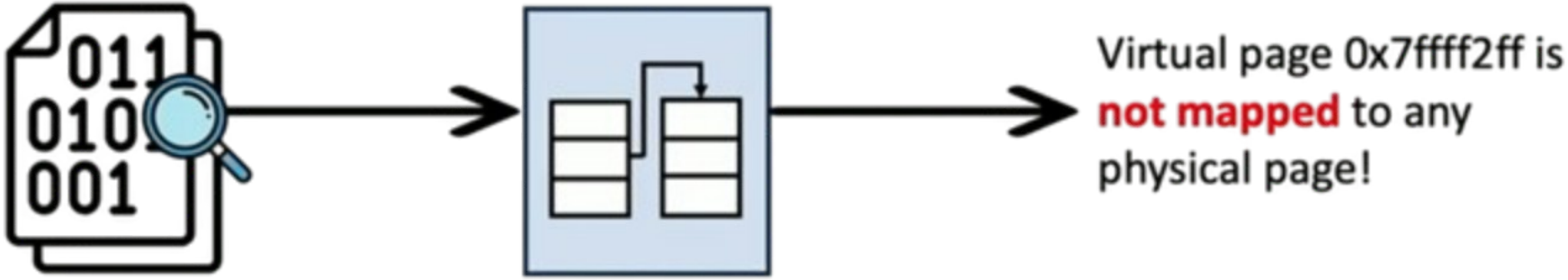
- Which physical address is 0x7ffff2fffd80 really mapped to?

GPU mem
(Dumped with DMA)

GPU page table

Demystifying GPU Memory & Out-of-Bounds Operations

- Which physical address is 0x7ffff2fffd80 really mapped to?



Demystifying GPU Memory & Out-of-Bounds Operations

Finding the memory location for local variables:

```
1  __global__ void local_arr(){
2      uint32_t arr[10];
3      for(int i = 0; i < 10; i++)
4          arr[i] = 0xdead0000+threadIdx.x;
5      uint32_t* ptr = arr;
6      printf("thread %u addr %p data %u\n",
7              threadIdx.x, ptr, ptr[0]);
8  }
9  int main(){
10     cuda_kernel<<<1,32>>>();
11     return 0;
12 }
```

Demystifying GPU Memory & Out-of-Bounds Operations

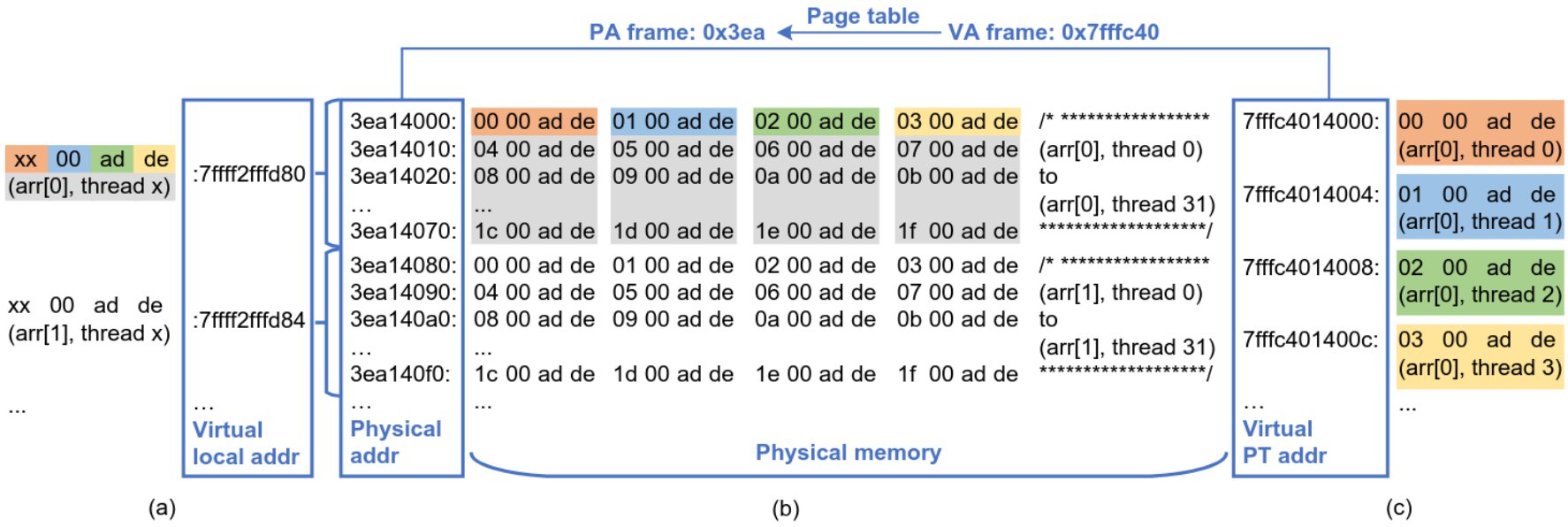


Figure 3: The mapping details of local memory: (a) the translation between virtual local addresses and physical addresses; (b) the layout of local memory (in device memory); (c) the translation between virtual PT addresses and physical addresses.

Demystifying GPU Memory & Out-of-Bounds Operations

- LDG/STG with a "real" local mem addr

```
/** R6.64:0x7fffc40xxxxx **/
LDG R8,[R6.64]
STG [R6.64],R8
```

Accesses local mem without errors!

One CUDA thread with an OOB operation in global memory may access or modify data in local memory of any active CUDA thread.

Demystifying GPU Memory & Out-of-Bounds Operations

Table 2: Summary of the buffer overflow problem in CUDA; ✓ means the OOB operation can affect this memory space, while ✗ means it cannot.

OOB		Global mem				Local mem				Shared mem			
		Same thread	Same thread block	Same kernel	Same program	Same thread	Same thread block	Same kernel	Same program	Same thread	Same thread block	Same kernel	Same program
Global mem	LDG/STG	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗
	LD/ST	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗
Local mem	LDL/STL	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
	LD/ST	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗
Shared mem	LDS/STS	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗
	LD/ST	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗

Threat Model

Attacker Goal: The primary objective is to compromise the integrity of the Deep Neural Network (DNN) to affect other users. Specifically:

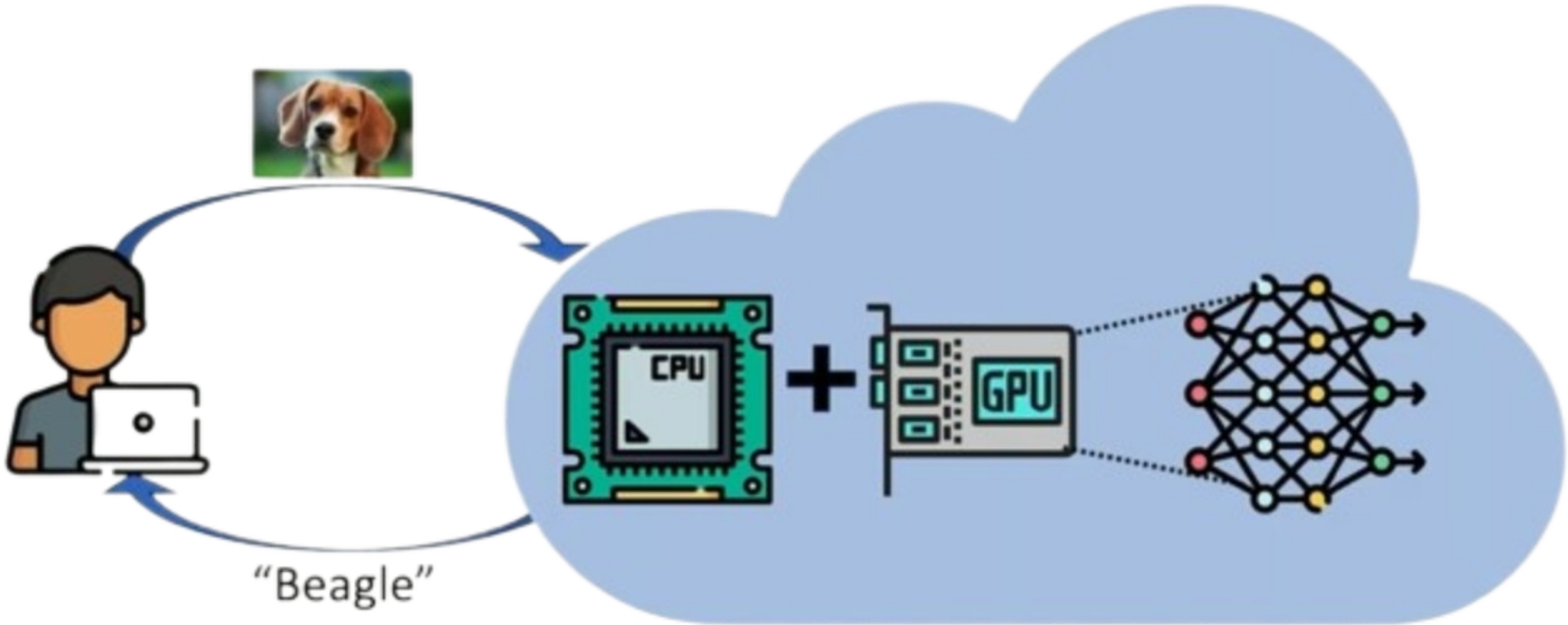
- The attacker seeks to exploit memory vulnerabilities to alter persistent data, such as model weights, stored in the GPU’s device memory.
- By modifying these weights during one request, the attacker ensures that all future inferences processed by the application remain compromised, leading to incorrect or malicious outputs for subsequent users.

Threat Model

Attacker Capabilities

- **Remote Access:** They can interact with the victim application by sending standard service requests over a network.
- **Vulnerability Exploitation:** They have the ability to craft malicious payloads that trigger buffer overflow vulnerabilities within the CUDA kernels running on the server’s NVIDIA GPU.
- **System Knowledge:** They possess detailed knowledge of the victim’s DNN model weight layout, allowing them to precisely target which parameters to overwrite.
- **Exploitation of Persistence:** They leverage the application's "always-on" memory management (where weights are loaded once to reduce latency) to ensure their changes stay in memory long after their malicious request has been processed.

Threat Model



Then where is this undisclosed memory location for the return address?

Then where is this undisclosed memory location for the return address?

Short answer…

Local Memory (AKA stack)

Return Address Corruption, Code injection & ROP attacks

Then where is this undisclosed memory location for the return address?

Short answer…

Local Memory (AKA stack)

Sometimes...

Return Address Corruption, Code injection & ROP attacks

```
/** Start of the func **/
/** RZ is always 0 **/
IADD3 R1, R1, -0x70, RZ ;
STL [R1+0x68], R25 ;
STL [R1+0x64], R24 ;
STL [R1+0x60], R23 ;
STL [R1+0x5c], R22 ;
STL [R1+0x58], R21 ;
STL [R1+0x54], R20 ;
STL [R1+0x50], R19 ;
STL [R1+0x4c], R18 ;
STL [R1+0x48], R17 ;
STL [R1+0x44], R16 ;
STL [R1+0x40], R2 ;
```

```
/** End of the func **/
LDL R2, [R1+0x40] ;
LDL R16, [R1+0x44] ;
LDL R17, [R1+0x48] ;
LDL R18, [R1+0x4c] ;
LDL R19, [R1+0x50] ;
LDL R20, [R1+0x54] ;
LDL R21, [R1+0x58] ;
LDL R22, [R1+0x5c] ;
LDL R23, [R1+0x60] ;
LDL R24, [R1+0x64] ;
LDL R25, [R1+0x68] ;
IADD3 R1, R1, 0x70, RZ ;
RET.ABS.NODEC R20 0x0 ;
```

Figure 5: Assembly code when entering/leaving the device function; the 64B local array in the device function is stored in `[R1]` to `[R1+0x3c]`.

Return Address Corruption, Code injection & ROP attacks

1. R1 register seems to act like the stack pointer despite being a general purpose register because there is no sp register in SASS

2. The ret call in the end uses R20 as an operand. So intuitively R20 should be related to the return address. Using CUDA GDB they saw the value of R20.64(which is R21 || R20) and it is in fact the same as the expected return address. Also R20 is retrieved from the stack before ret so the return address should be somewhere stored in the stack.

Return Address Corruption, Code injection & ROP attacks

2 Main reasons for the return address to be in stack:

- The device function is recursive.
- The device function has a substantial number of local variables, resulting in insufficient registers (i.e., register spill-out).

Code Injection?

Return Address Corruption, Code injection & ROP attacks

All pages are executable

- Including local memory (stack), global memory…
- No NX bit in PTE

All pages are writable

- There is a Read-only bit in PTE, but its never set
- After setting this bit for code pages, writing causes an error

What about ROP?

Return Address Corruption, Code injection & ROP attacks

- libcuda: a close-sourced CUDA driver API library.
- Part of this code is loaded for every CUDA context.
- Contains around 50 ROP gadgets.
- Other common CUDA libs, such as libcuDNN, libcuBLAS, do not have ROP gadgets.

Memory write gadget

```
ST.E.64 [R28.64], R4 ;
BSYNC B7 ;
LDL R0, [R1] ;
BMOV.32 B6, R27 ;
LDL R20, [R1+0x18] ;
LDL R21, [R1+0x1c] ;
LDL R2, [R1+0x4] ;
LDL R16, [R1+0x8] ;
LDL R17, [R1+0xc] ;
…
LDL R29, [R1+0x3c] ;
IADD3 R1, R1, 0x40, RZ ;
BMOV.32 B7, R0 ;
RET.ABS.NODEC R20 0x0 ;
```

Evaluation: attacks on Deep learning

Added a buffer overflow vulnerability to 4 vision models implemented using PyTorch

- ResNet-18
- ResNet-50
- VGGNet
- Vision Transformer

DNN inference application hosted in a virtual machine on a cloud system with a server-grade GPU. The GPU was virtualized using NVIDIA's virtual GPU (vGPU) technology. **vGPU does not support CUDA ASLR**

Evaluation: attacks on Deep learning

Table 3: The DNN inference accuracy with the weight modification attacks (only for weights in the first layer).

DNN model	Clean acc.	Weight modification (controlled)				Weight modification (uncontrolled)			
		10%	20%	50%	100%	10%	20%	50%	100%
ResNet-18 (CIFAR-10)	87.37%	10.00%	10.00%	10.00%	10.00%	87.14%	87.32%	81.65%	10.83%
VGG-19 (CIFAR-10)	83.56%	13.46%	13.46%	11.19%	9.66%	74.90%	62.77%	59.01%	10.00%
ResNet-50 (ImageNet-1K)	84.97%	58.21%	55.21%	54.75%	44.35%	84.93%	84.27%	80.64%	65.33%
ViT (ImageNet-1K)	93.64%	0.09%	0.08%	0.12%	0.09%	92.78%	90.56%	90.33%	88.54%

DNN Model	Clean acc.	Uncontrolled Weight modification			
		10%	20%	50%	100%
Flan-T5-Small	29.5%	29.2%	28.6%	28.3%	28.2%
Flan-T5-Base	34.2%	33.2%	32.6%	28.9%	29.5%
Flan-T5-Large	42.0%	41.3%	40.5%	39.1%	35.0%

Proposed Mitigations & Conclusion

- **ASLR and PIE**: CUDA does support ASLR for both code and data memory and its on by default, but the CUDA driver API library (which contains ROP gadgets) is always loaded to a fixed address even with full randomization on.
- **OOB detection tools:** Tools to statically/dynamically detect buffer overflow errors in GPU programs. NVIDIA Compute Sanitizer . It introduces too much overhead. CuCatch compile-time tool, less overhead. Both tools cannot reach 100% detection rate
- **Stack cookies:** No implementation from NVIDIA also known to be by-passable.
- **W^X policy:** There is already a read-only bit in PTE. JUST USE IT!! Also add an Executable bit.

Thank you for listening!

Any questions?