NVIDIA

cuCatch: A Debugging Tool for Efficiently Catching Memory Safety Violations in CUDA Applications

Mohamed Tarek Ibn Ziad, Sana Damani, Aamer Jaleel, Stephen W. Keckler, Mark Stephenson
NVIDIA | PLDI 2023

GPUs for Accelerating General-Purpose Software

Large Language Models (LLMs)

https://developer.nvidia.com/blog/efficiently-scale-llm-training-across-a-large-gpu-cluster-with-alpa-and-ray/

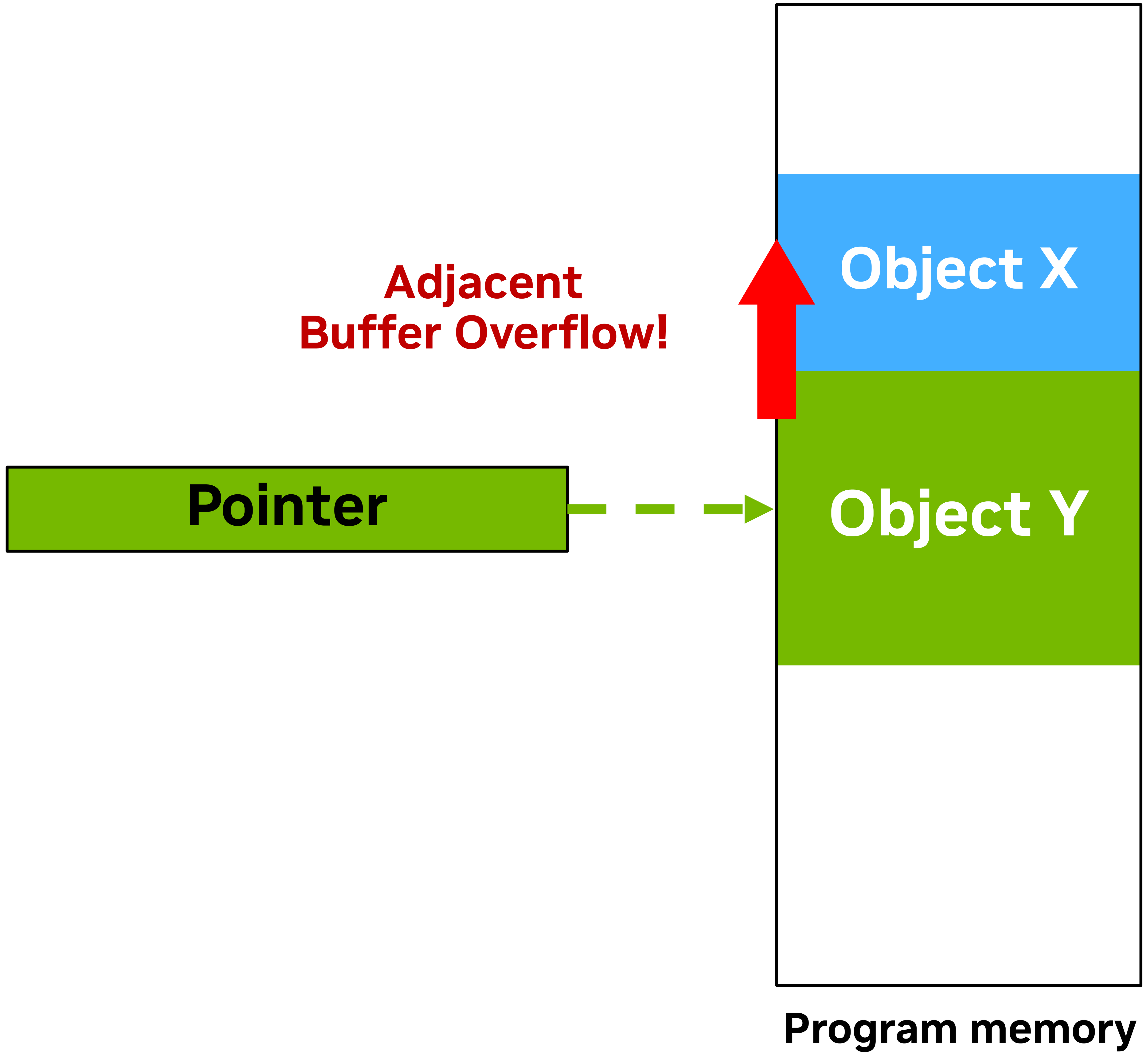
Autonomous Driving

https://developer.nvidia.com/blog/developing-an-end-to-end-auto-labeling-pipeline-for-autonomous-vehicle-perception/

Robotics

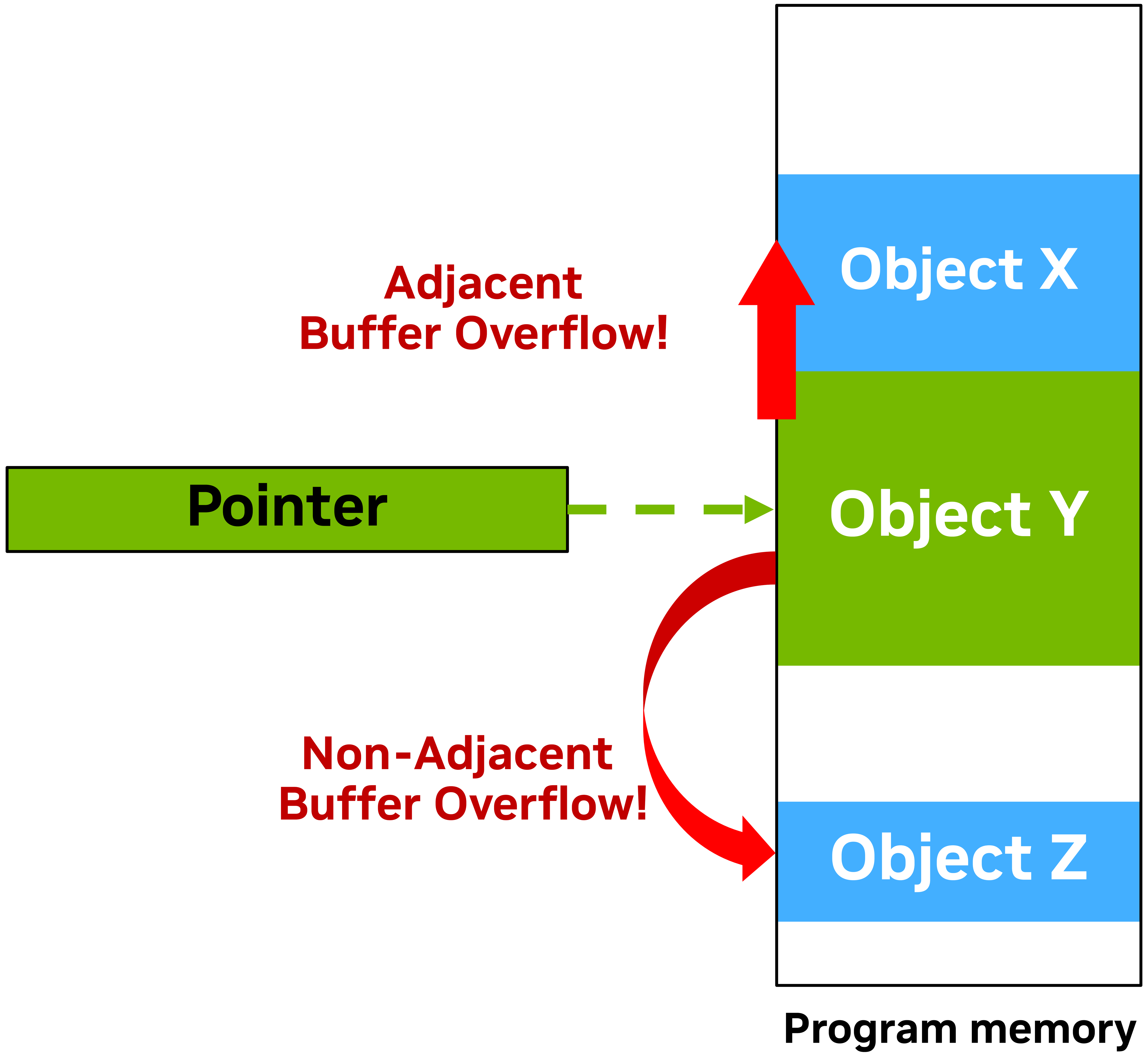
https://blogs.nvidia.com/blog/2023/05/28/isaac-amr-nova-orin-autonomous-mobile-robots/

Memory Safety Errors



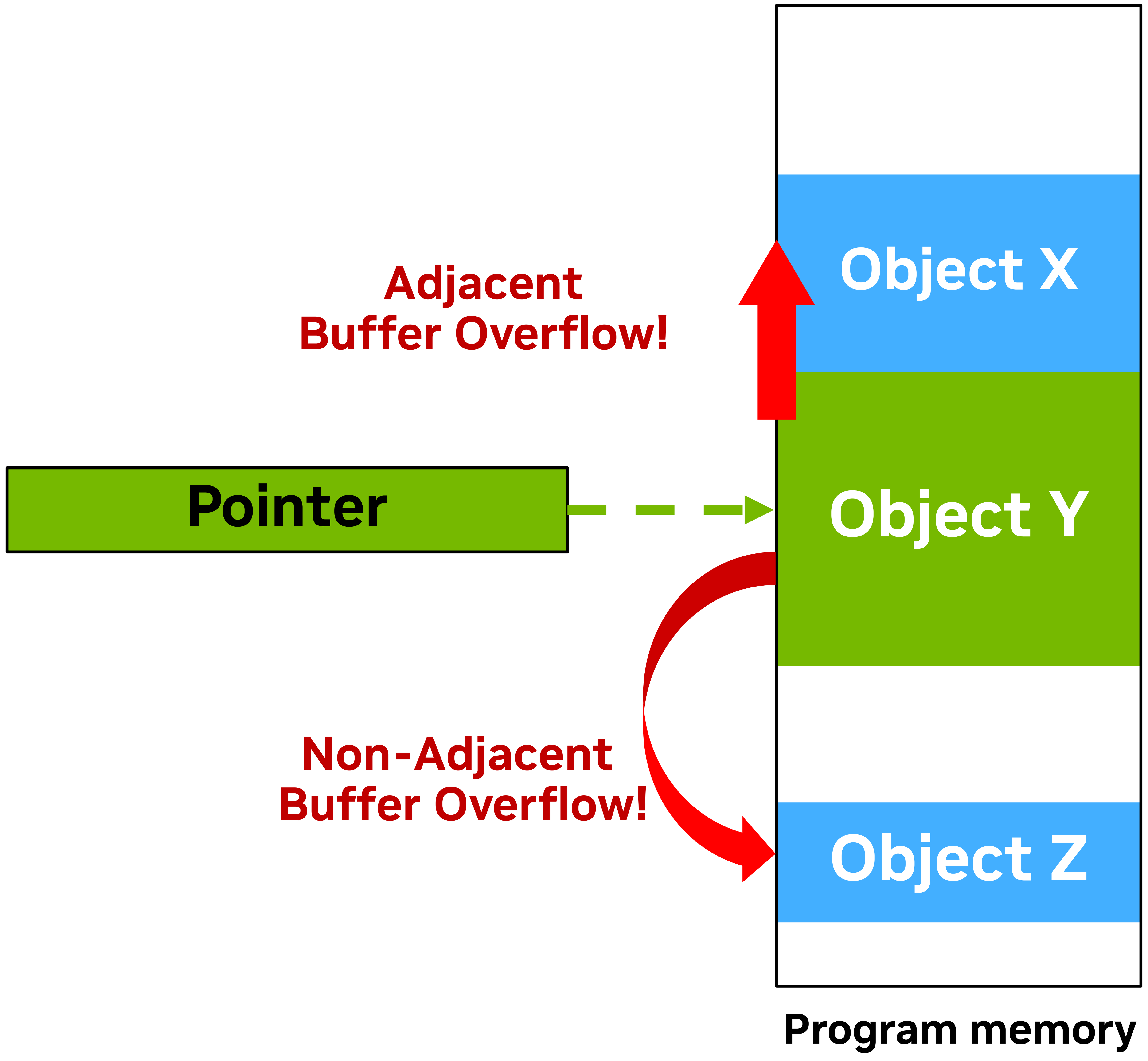
Spatial memory safety errors

Memory Safety Errors

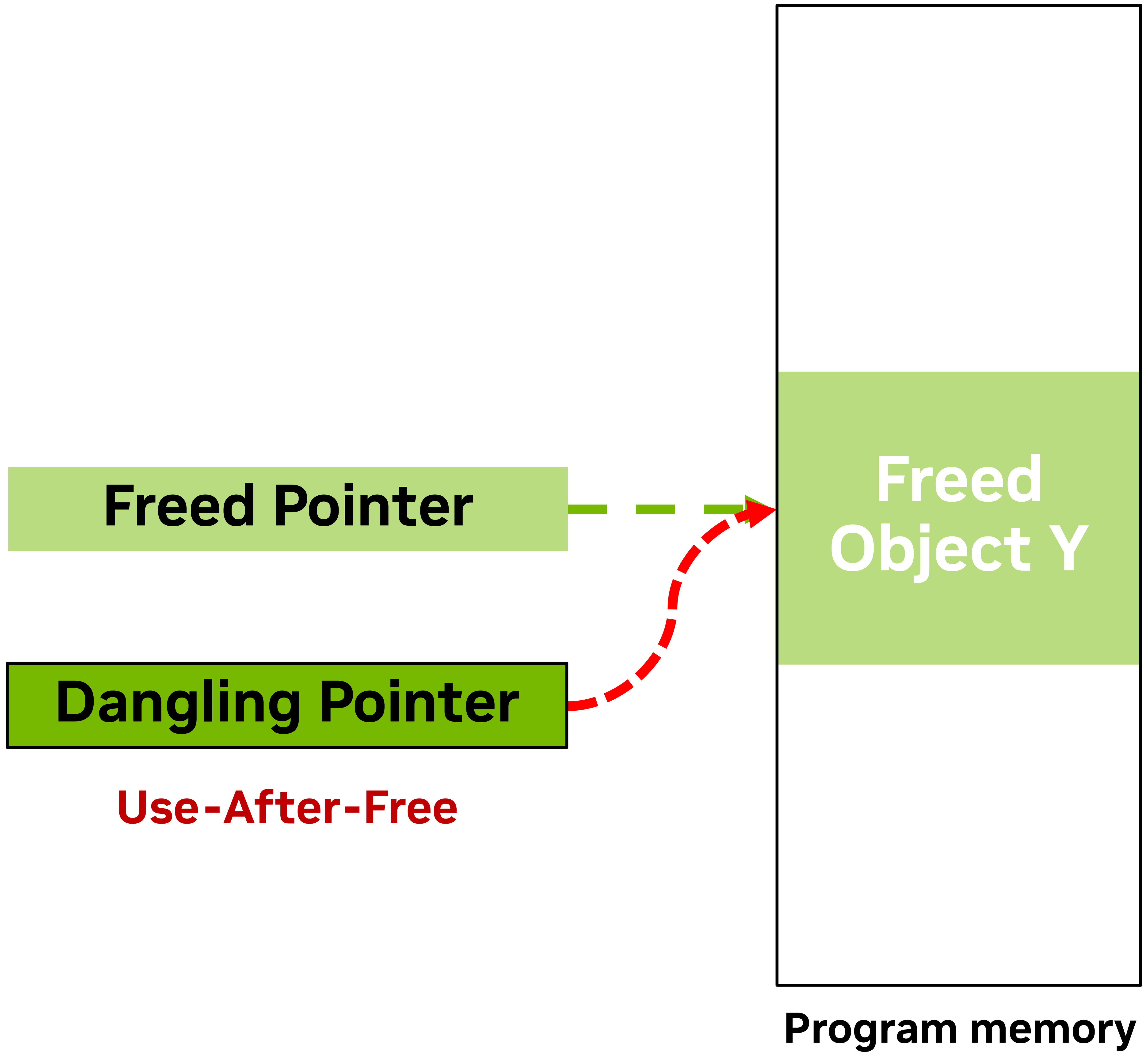


Spatial memory safety errors

Memory Safety Errors

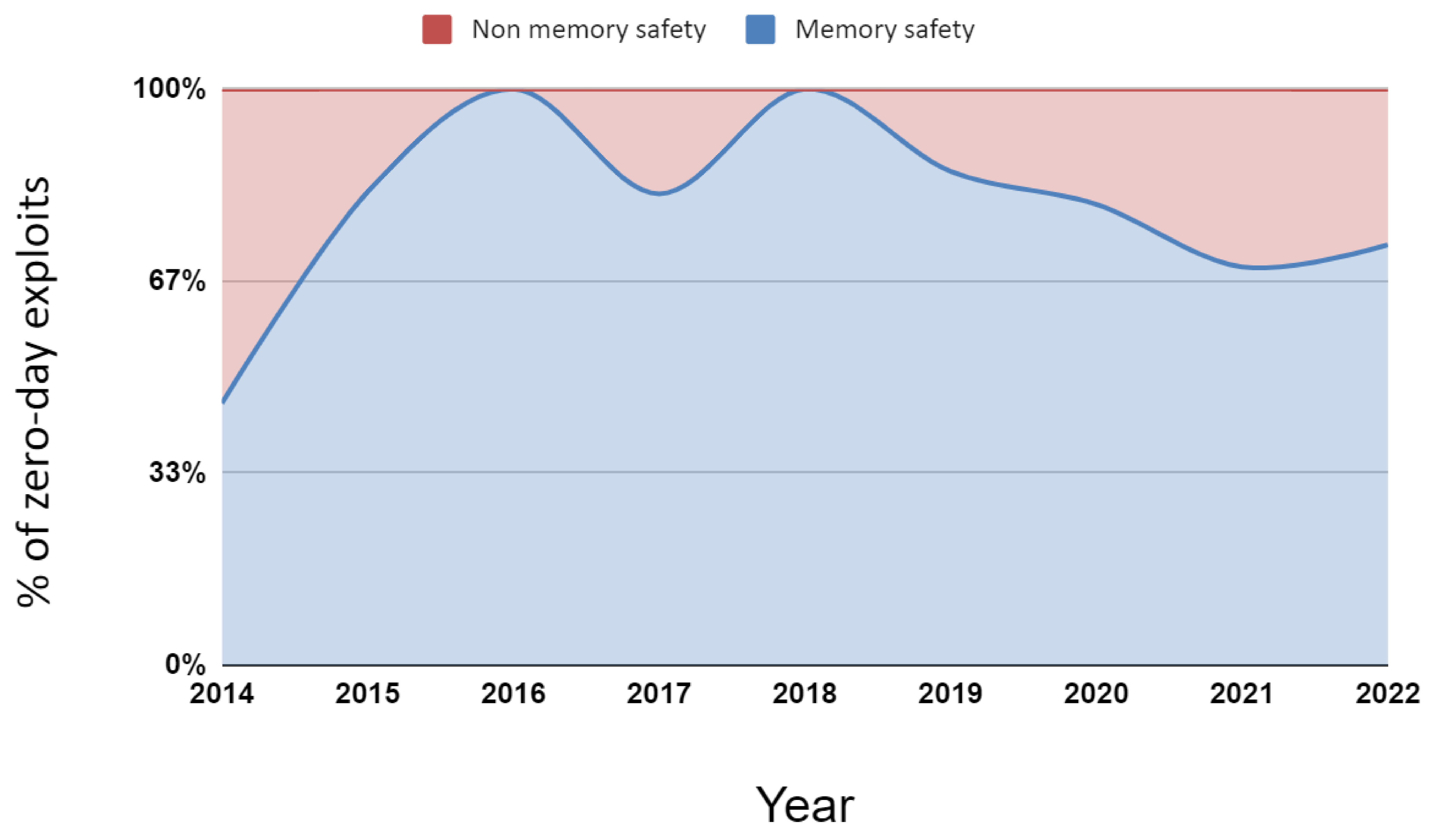


Spatial memory safety errors



Temporal memory safety errors

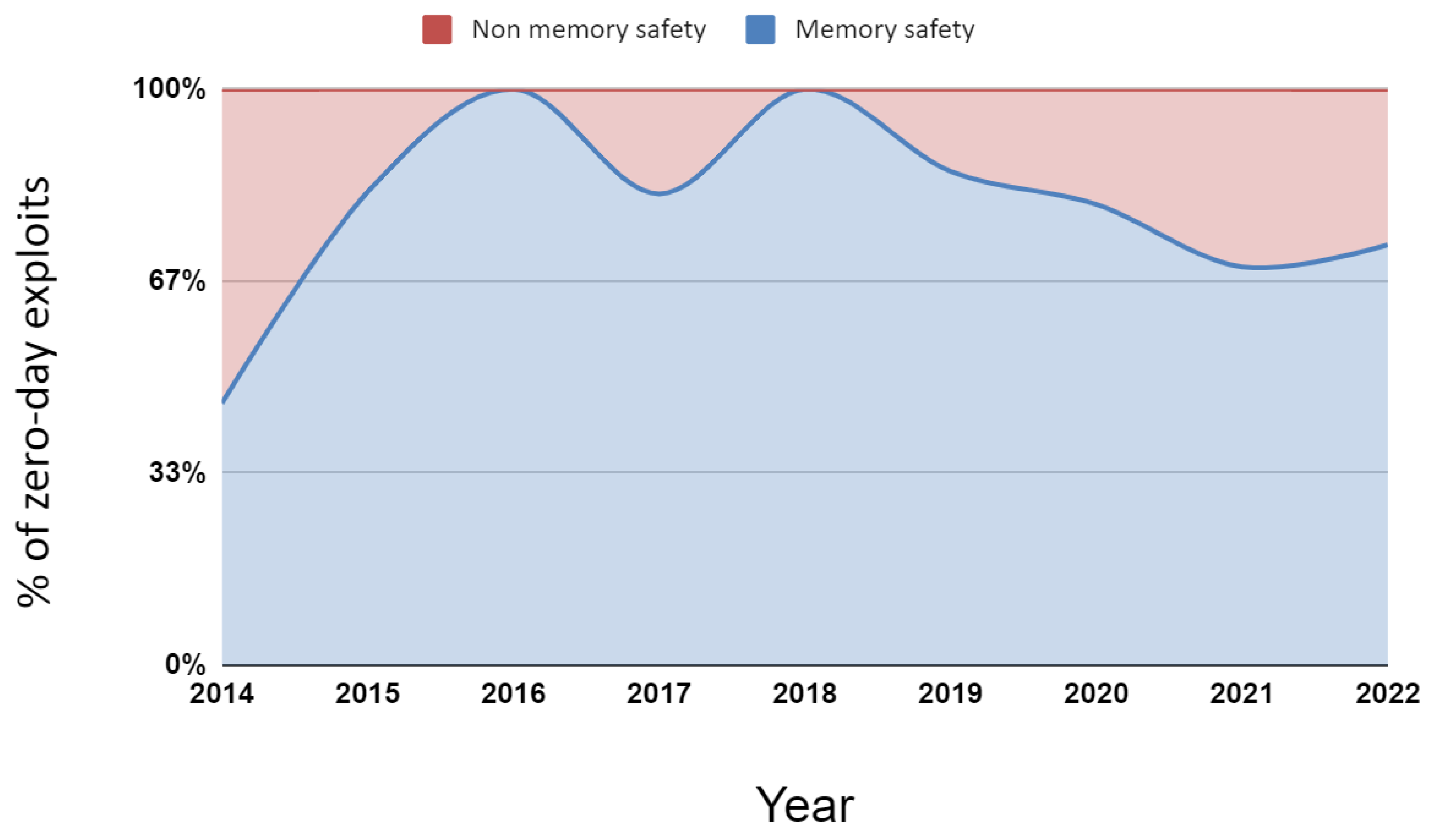
CPUs and GPUs are Vulnerable to Memory Safety Errors



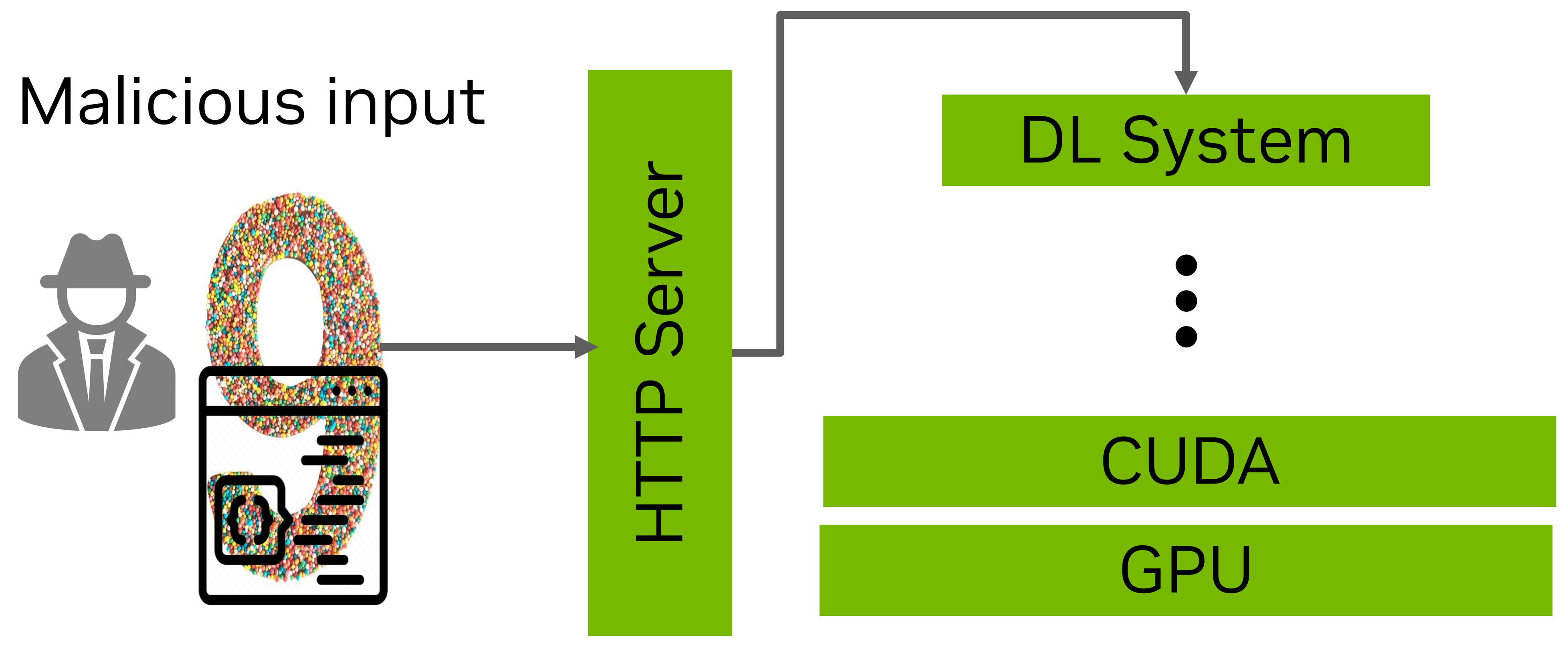
% of Zero-day "in the wild" exploits from 2014-2022

Source: *Google Project Zero, 0day "In the Wild" spreadsheet.*
Last updated: January 2023

CPUs and GPUs are Vulnerable to Memory Safety Errors



% of Zero-day “in the wild” exploits from 2014-2022

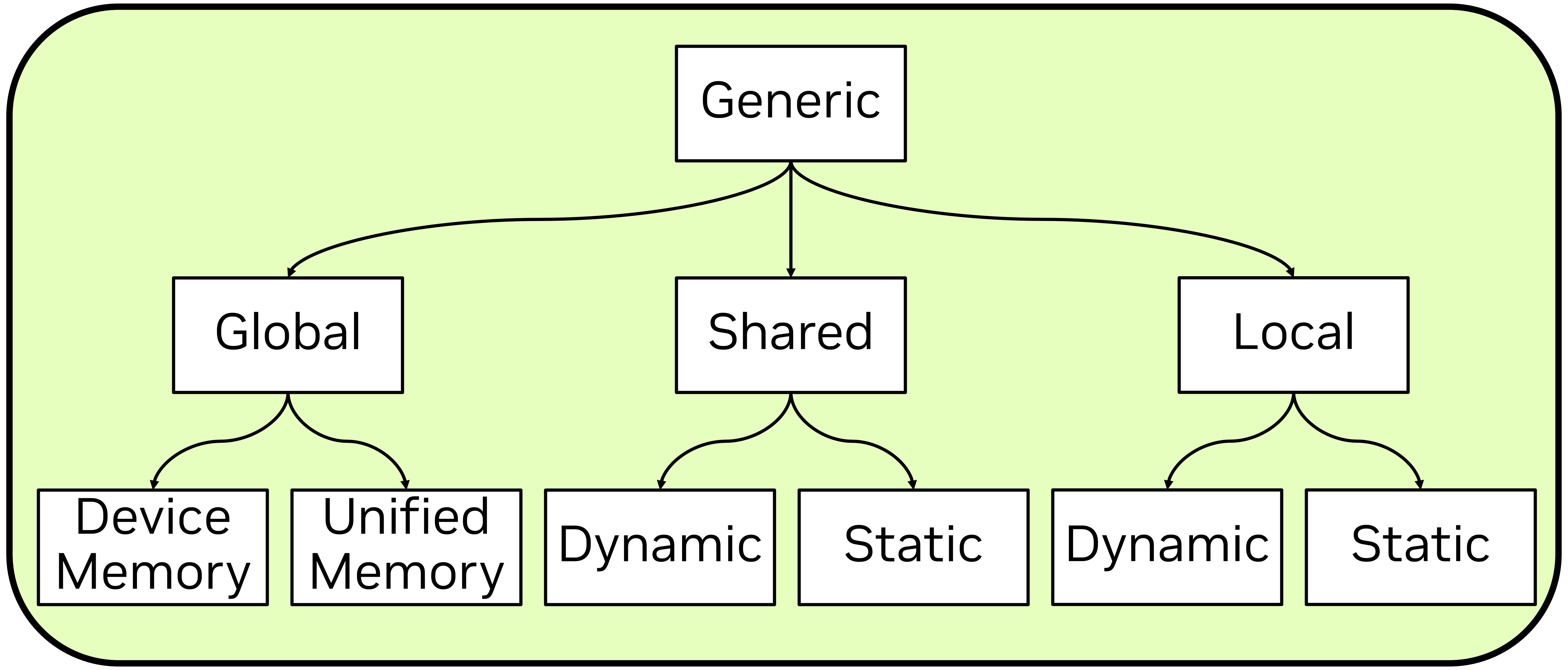


Exploiting a memory safety error to hijack the GPU

Source: *Google Project Zero, 0day "In the Wild" spreadsheet. Last updated: January 2023*

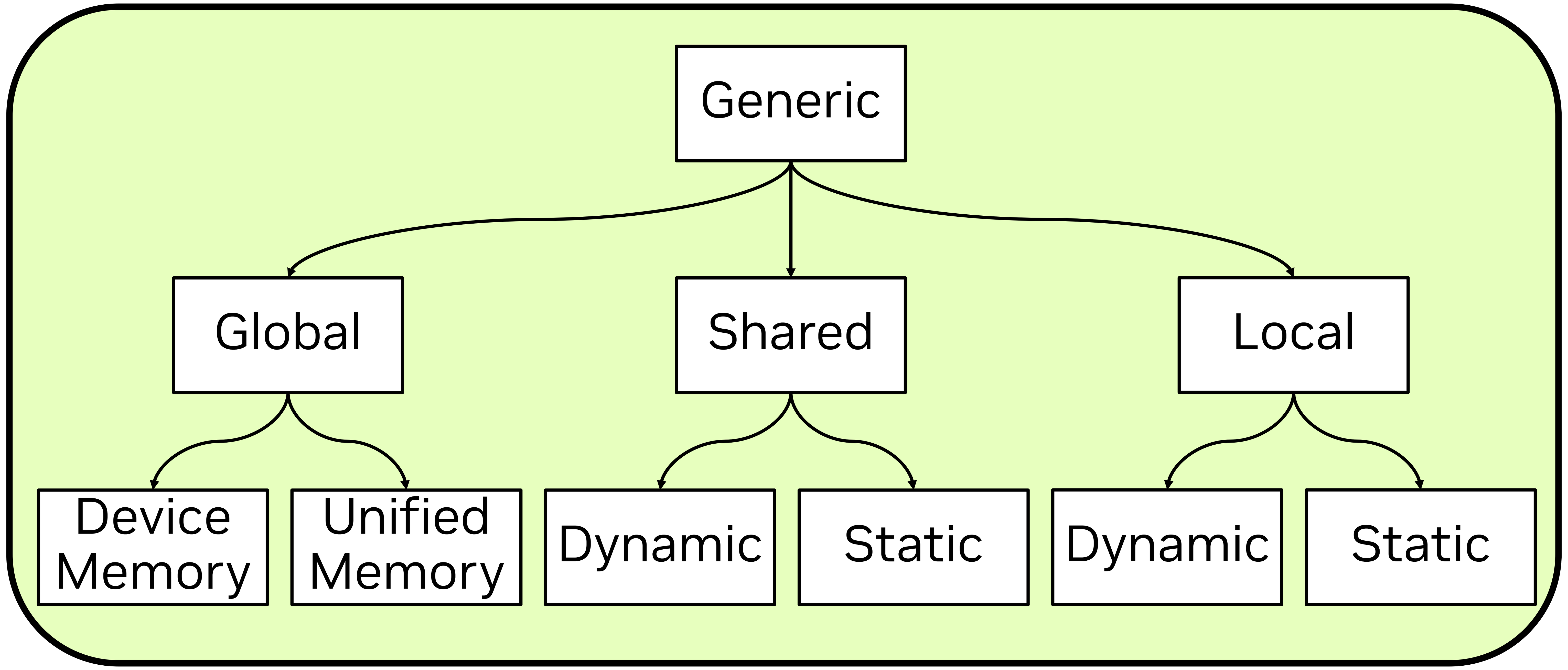
Source: *Park et al., Mind Control Attack: Undermining Deep Learning with GPU Memory Exploitation, Computers & Security, March 2021.*

Adopting CPU-Based Memory Safety Solutions to GPUs is Challenging

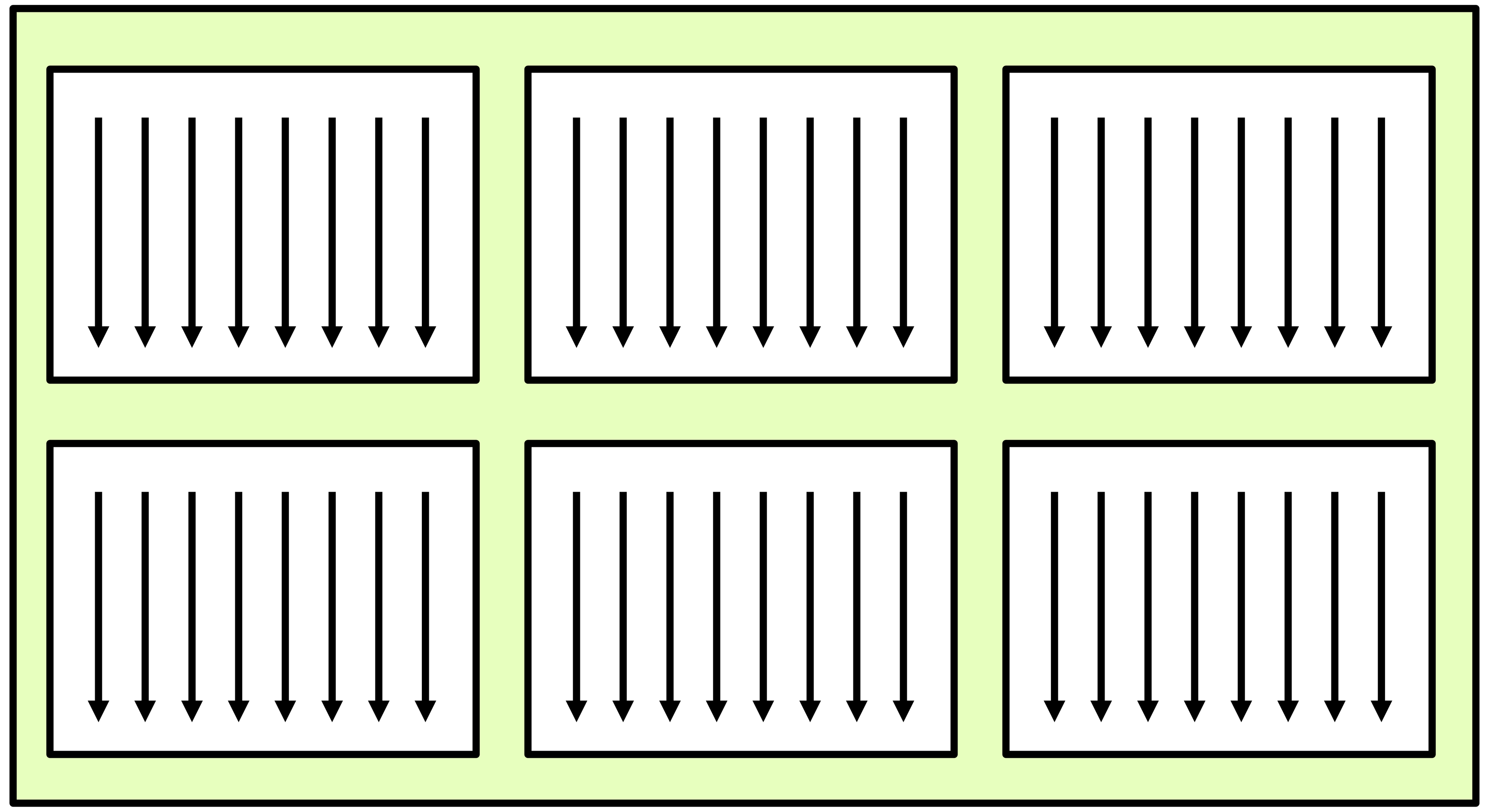


Distinct GPU memory spaces

Adopting CPU-Based Memory Safety Solutions to GPUs is Challenging



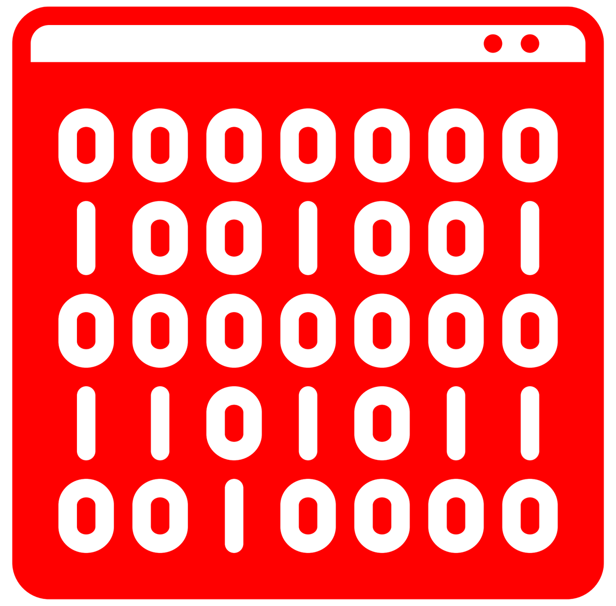
Distinct GPU memory spaces



Massive GPU multi-threading

Existing GPU-based Solutions are NOT Practical

Dynamic Binary Instrumentation



+ No SW/HW Changes
- High Runtime Overheads
- Low Detection Coverage

NVIDIA's Compute Sanitizer

Existing GPU-based Solutions are NOT Practical

Dynamic Binary Instrumentation

+ No SW/HW Changes
- High Runtime Overheads
- Low Detection Coverage

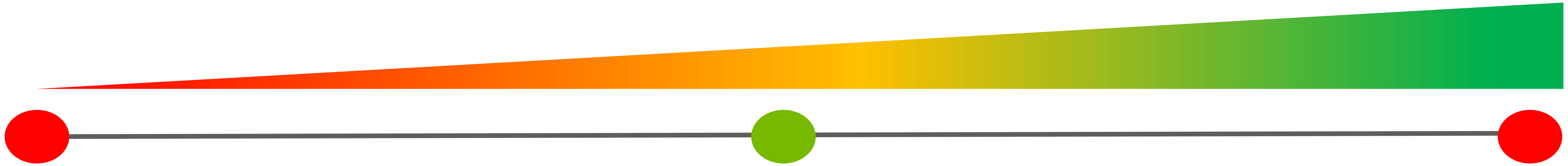
NVIDIA's Compute Sanitizer

Hardware Acceleration

- Requires HW Changes
+ Low Runtime Overheads
+ High Detection Coverage

GPUShield [ISCA 2022]

Existing GPU-based Solutions are NOT Practical



Dynamic Binary Instrumentation

+ No SW/HW Changes
- High Runtime Overheads
- Low Detection Coverage

NVIDIA's Compute Sanitizer

Compiler-based Instrumentation

+ No SW/HW Changes
+ Low Runtime Overheads
+ High Detection Coverage

cuCatch

Hardware Acceleration

- Requires HW Changes
+ Low Runtime Overheads
+ High Detection Coverage

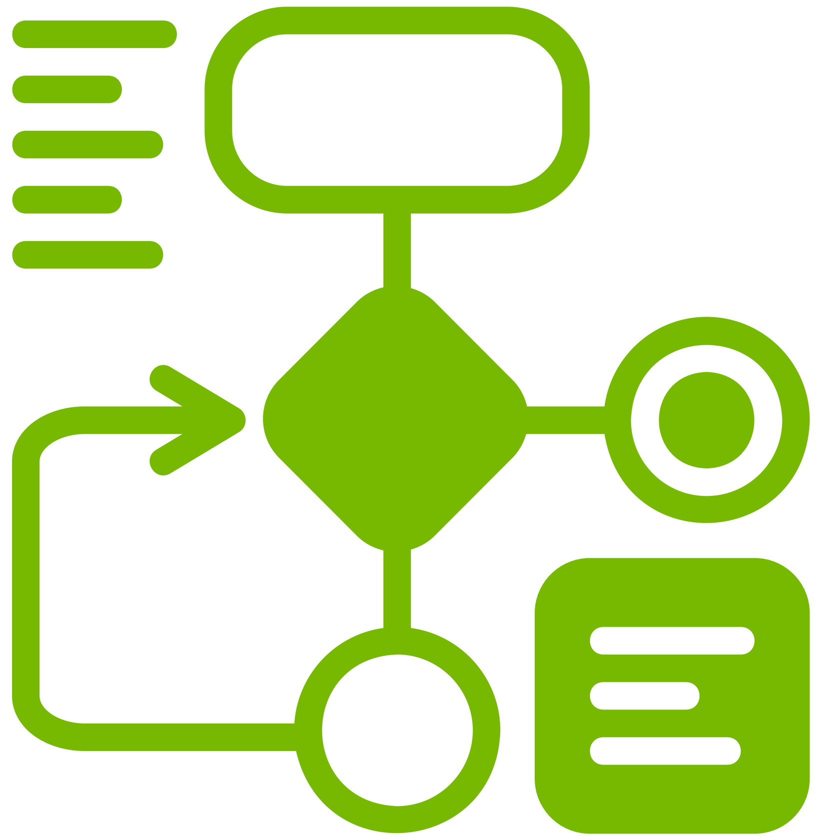
GPUShield [ISCA 2022]

- cuCatch Overview
- Error Detection Coverage
- Evaluation Results
- Conclusion

- **cuCatch Overview**
- Error Detection Coverage
- Evaluation Results
- Conclusion

cuCatch Overview

Two Main Components

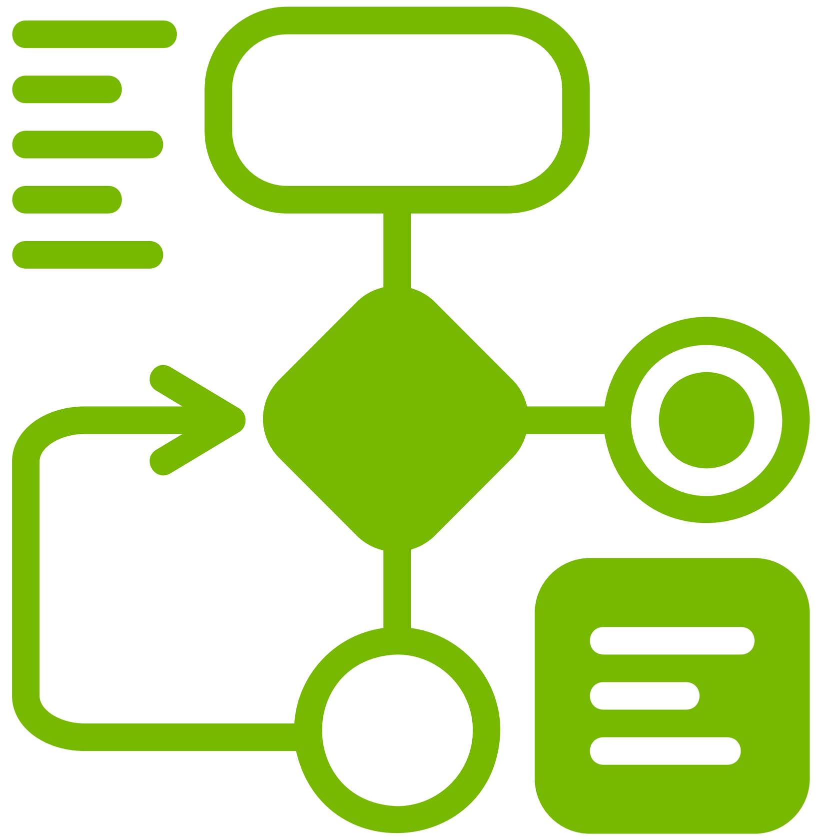


Novel Memory Safety Algorithm

Shadow Tagged Base & Bounds

cuCatch Overview

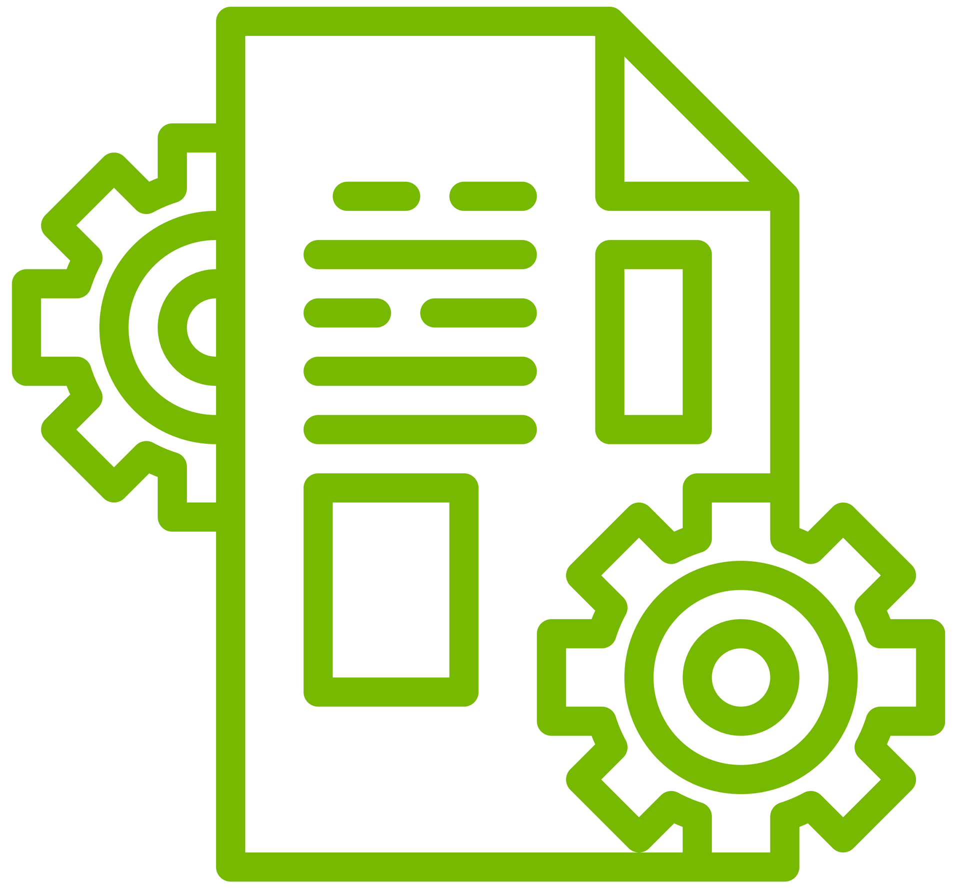
Two Main Components



Novel Memory Safety Algorithm

Shadow Tagged Base & Bounds

+

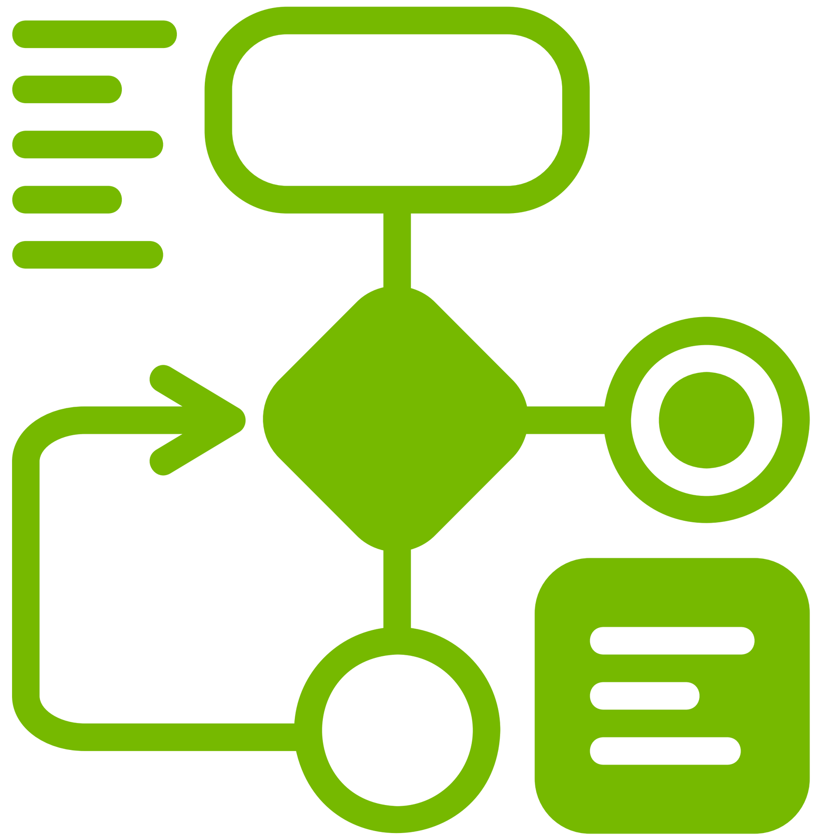


Novel Compile-Time Analysis

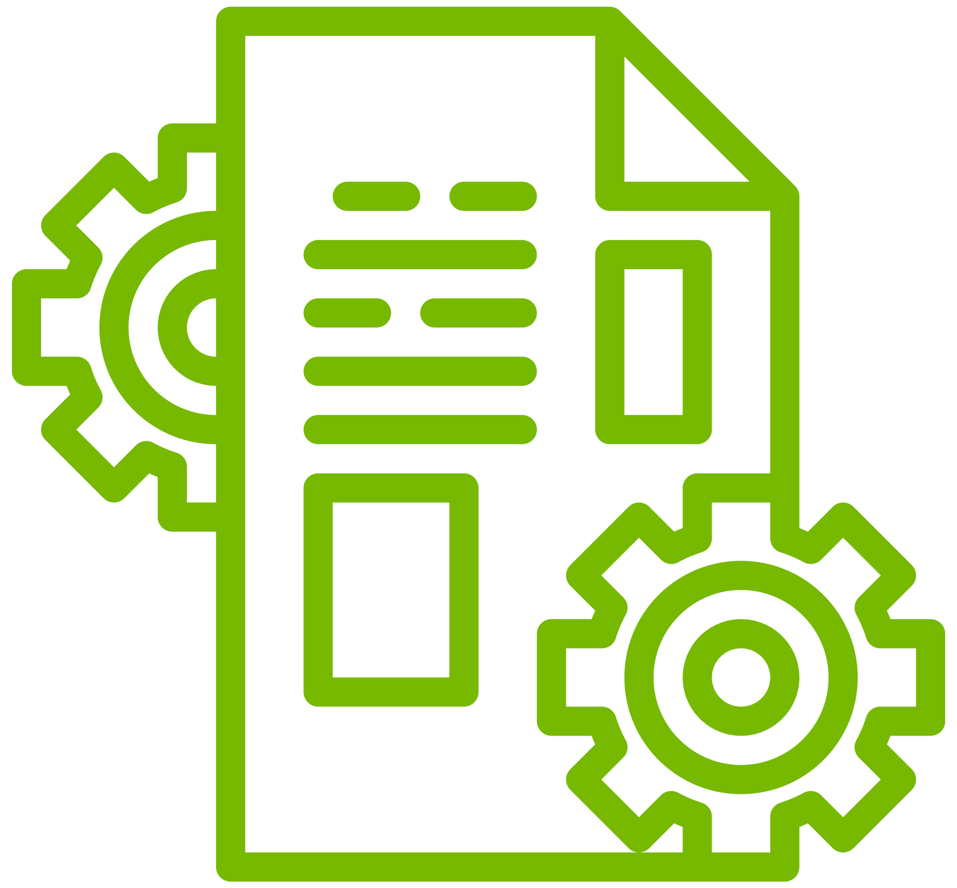
Base-Pointer Analysis

cuCatch Overview

Two Main Components



+



Novel Memory Safety Algorithm
Shadow Tagged Base & Bounds

Novel Compile-Time Analysis
Base-Pointer Analysis

Goal: construct “fat” pointers by eagerly retrieving allocation base and bound information without changing the application binary interface

cuCatch Algorithm: Shadow Tagged Base & Bounds

cuCatch Algorithm: Shadow Tagged Base & Bounds

Program memory

```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;

while(condition)
{
  t = *p;

  k = foo(t);

  p = p + k;

}

cudaFree(x);
.
.
.
p[0] = 0;
```

cuCatch Algorithm: Shadow Tagged Base & Bounds

Program memory

```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;

while(condition)
{
  t = *p;

  k = foo(t);

  p = p + k;

}

cudaFree(x);
.
.
.
p[0] = 0;
```


NVIDIA

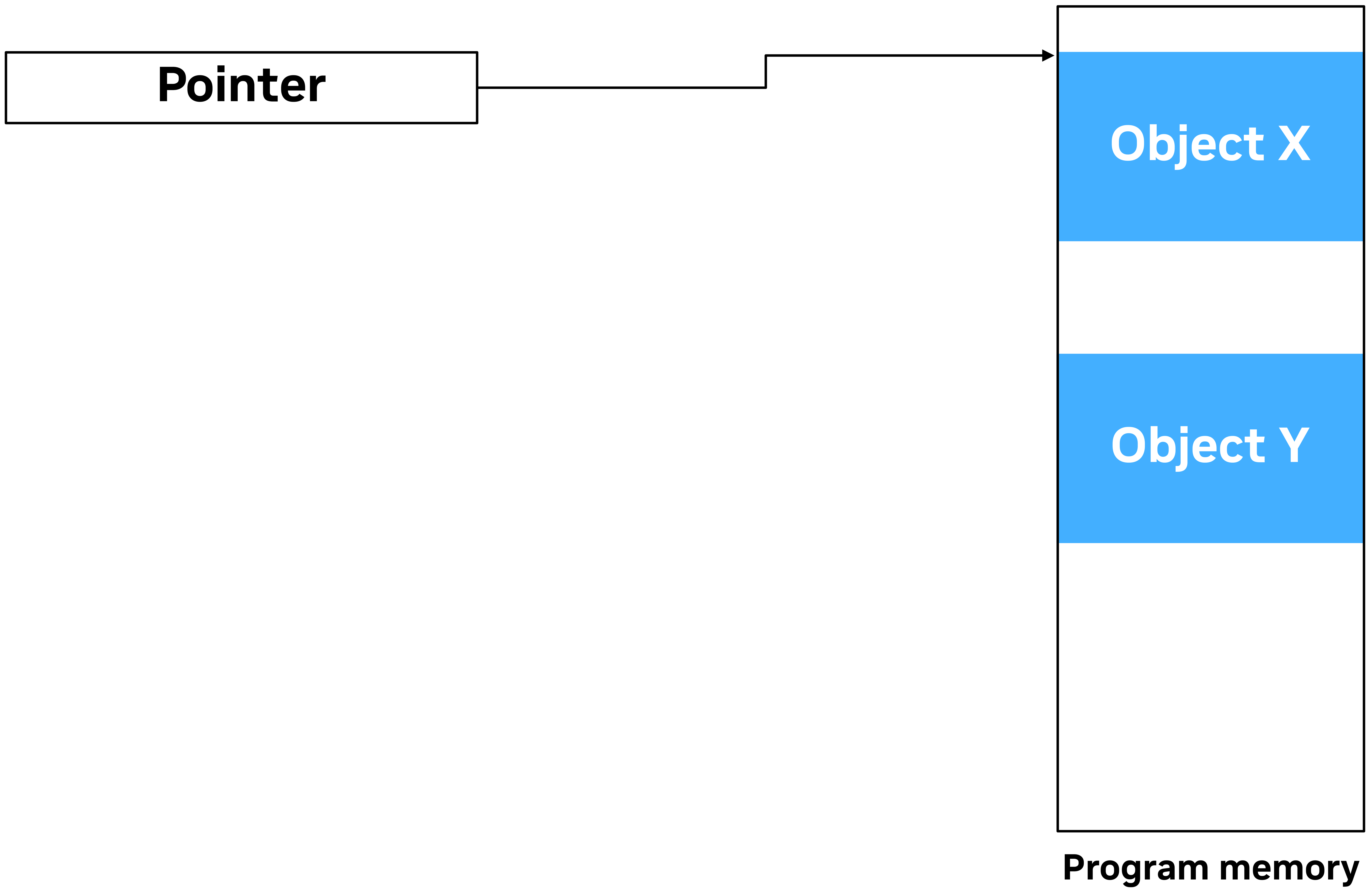
cuCatch Algorithm: Shadow Tagged Base & Bounds

Program memory

```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;

while(condition)
{

   t = *p;

   k = foo(t);

   p = p + k;

}

cudaFree(x);
.
.
.
p[0] = 0;
```

cuCatch Algorithm: Shadow Tagged Base & Bounds

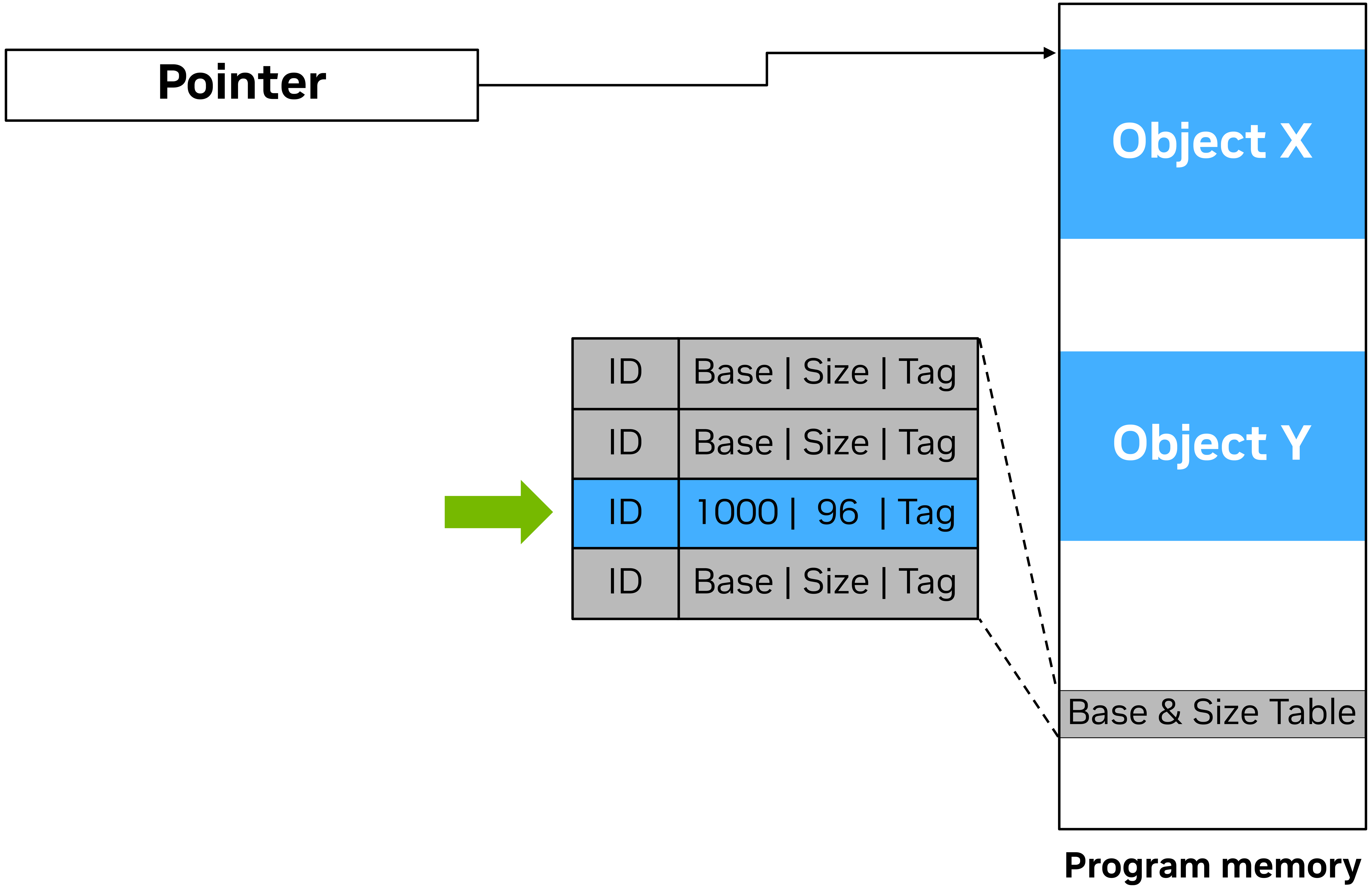
Memory Allocation




```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;

while(condition)
{
  t = *p;
  k = foo(t);
  p = p + k;
}

cudaFree(x);
.
.
.
p[0] = 0;
```

cuCatch Algorithm: Shadow Tagged Base & Bounds

Memory Allocation: Runtime Support

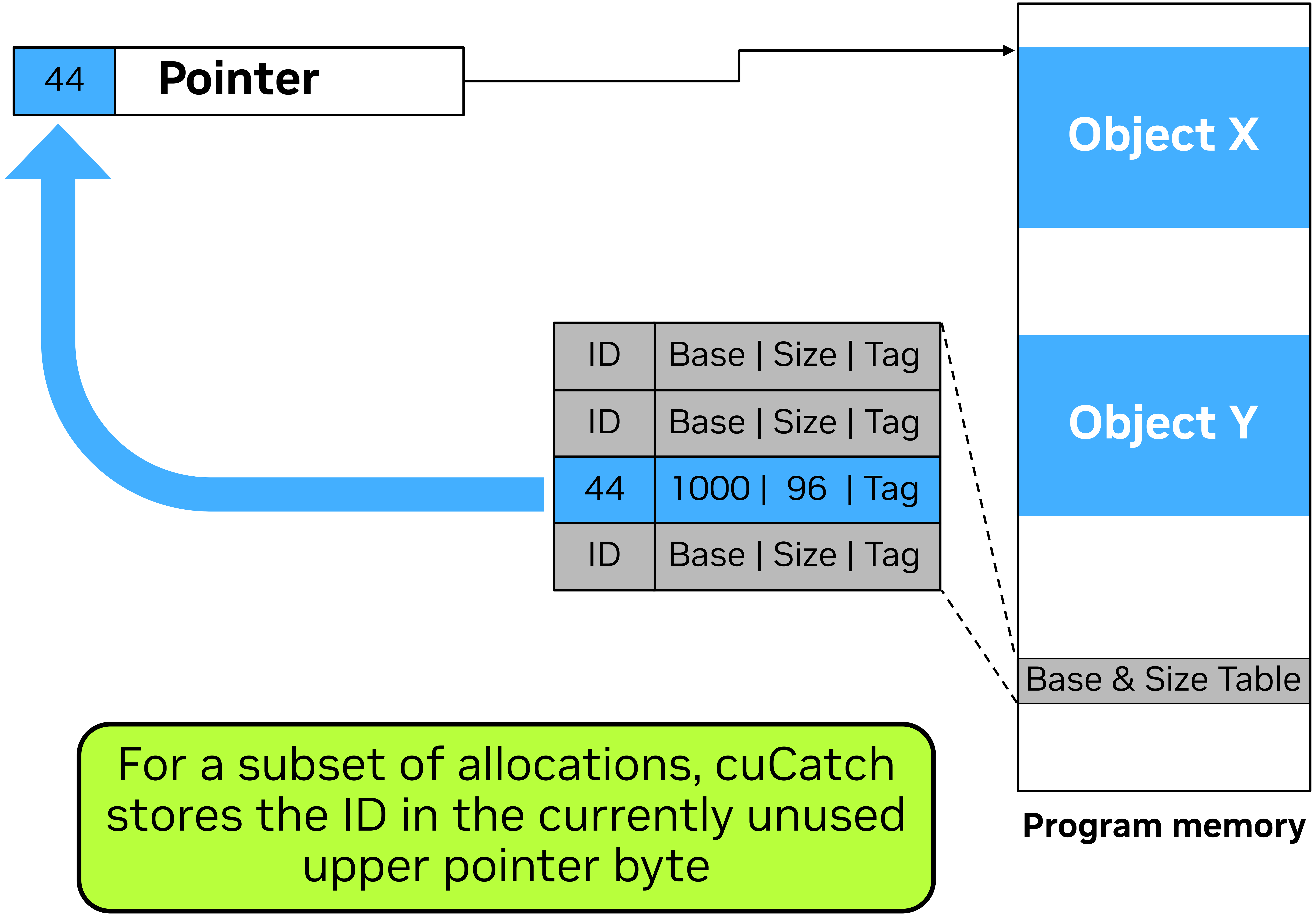


ID	Base | Size | Tag
ID	Base | Size | Tag
ID	1000 | 96 | Tag
ID	Base | Size | Tag

```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;

while(condition)
{
   t = *p;

   k = foo(t);

   p = p + k;

}

cudaFree(x);
.
.
.
p[0] = 0;
```

cuCatch Algorithm: Shadow Tagged Base & Bounds

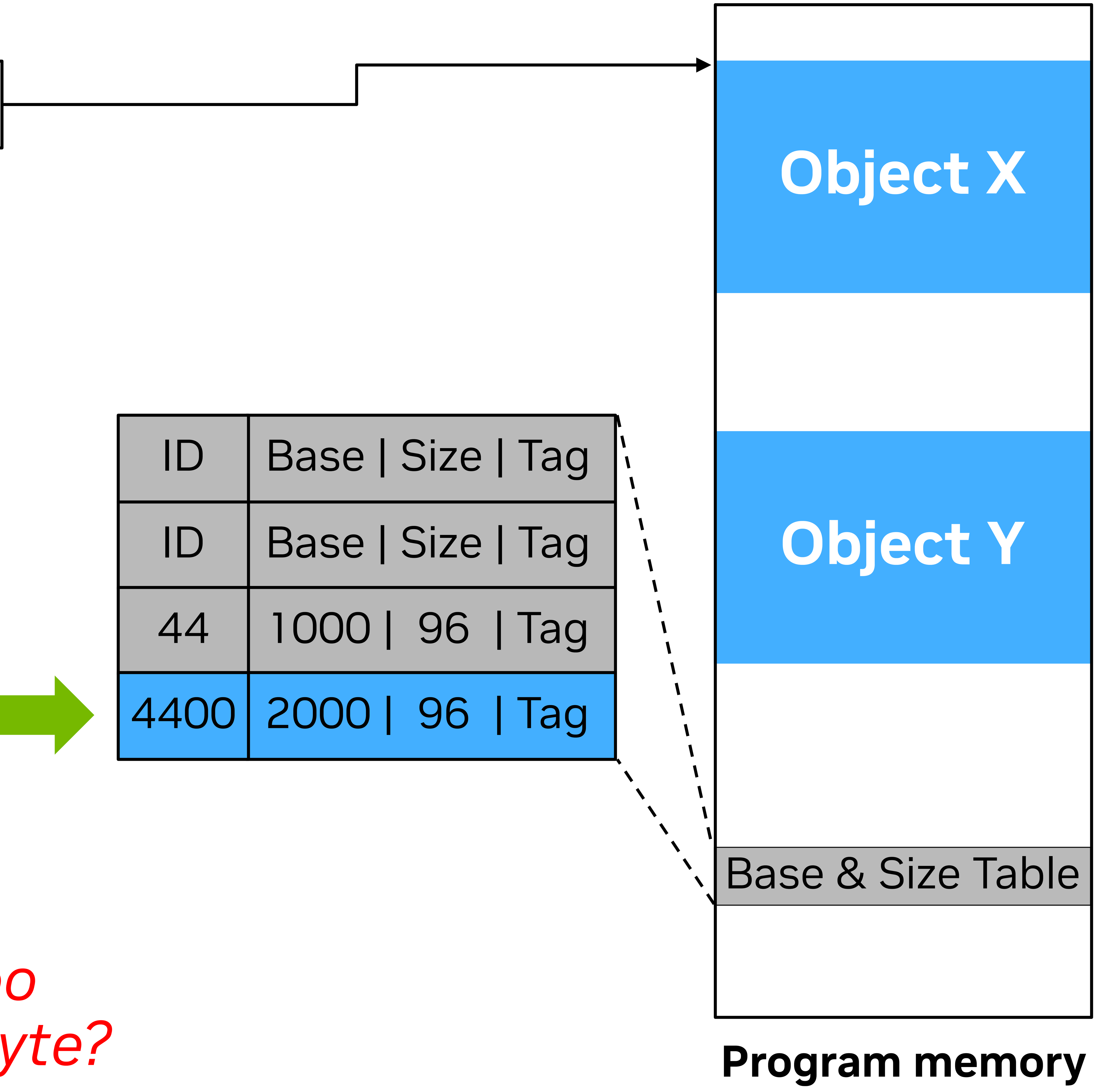
Memory Allocation: Runtime Support



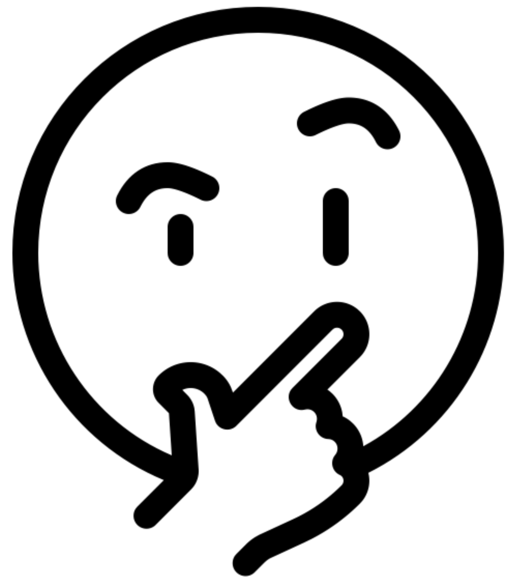

```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;

while(condition)
{
  t = *p;

  k = foo(t);

  p = p + k;

}

cudaFree(x);
.
.
.
p[0] = 0;
```

cuCatch Algorithm: Shadow Tagged Base & Bounds

Memory Allocation: Runtime Support



ID	Base | Size | Tag
ID	Base | Size | Tag
44	1000 | 96 | Tag
4400	2000 | 96 | Tag

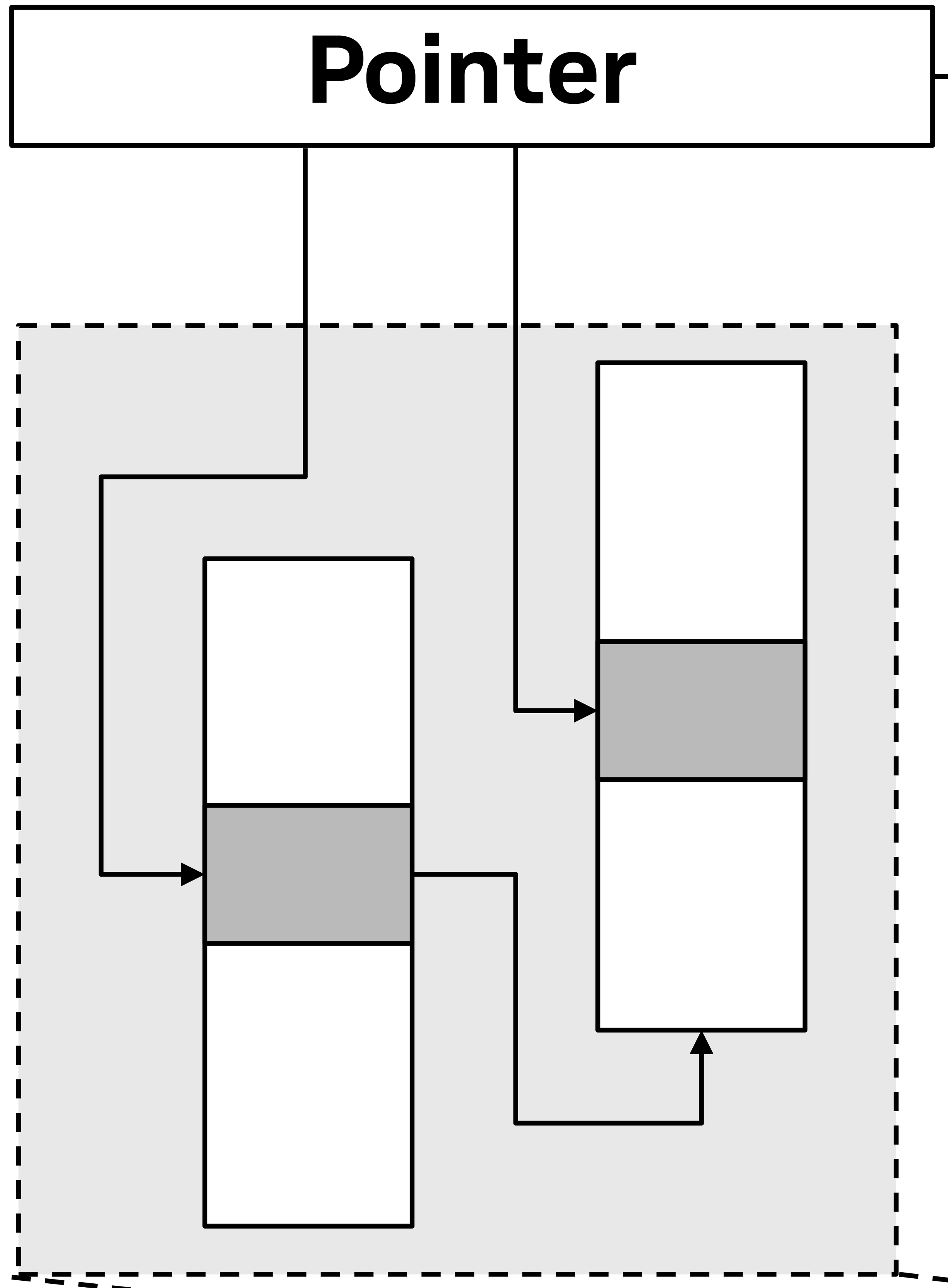


What if the ID is too large for a single byte?

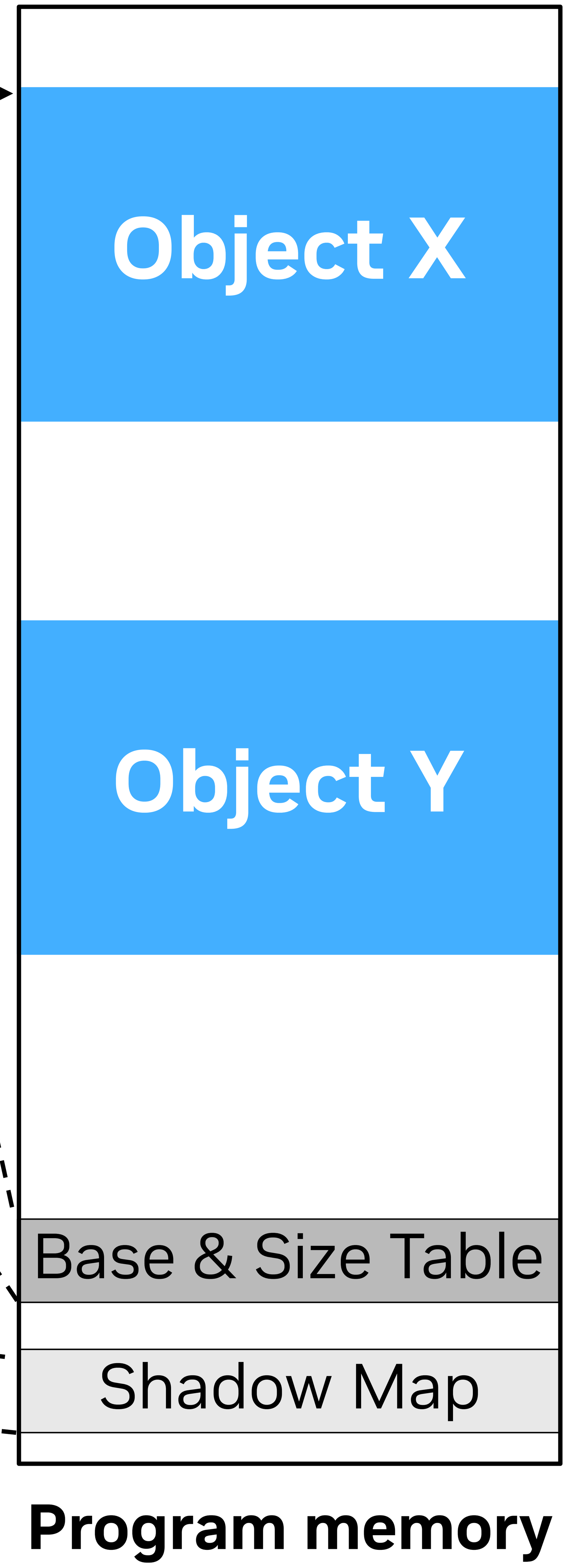
```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;

while(condition)
{

  t = *p;

  k = foo(t);

  p = p + k;

}

cudaFree(x);
.
.
.
p[0] = 0;
```

cuCatch Algorithm: Shadow Tagged Base & Bounds

Memory Allocation: Runtime Support



ID	Base | Size | Tag
ID	Base | Size | Tag
44	1000 | 96 | Tag
4400	2000 | 96 | Tag

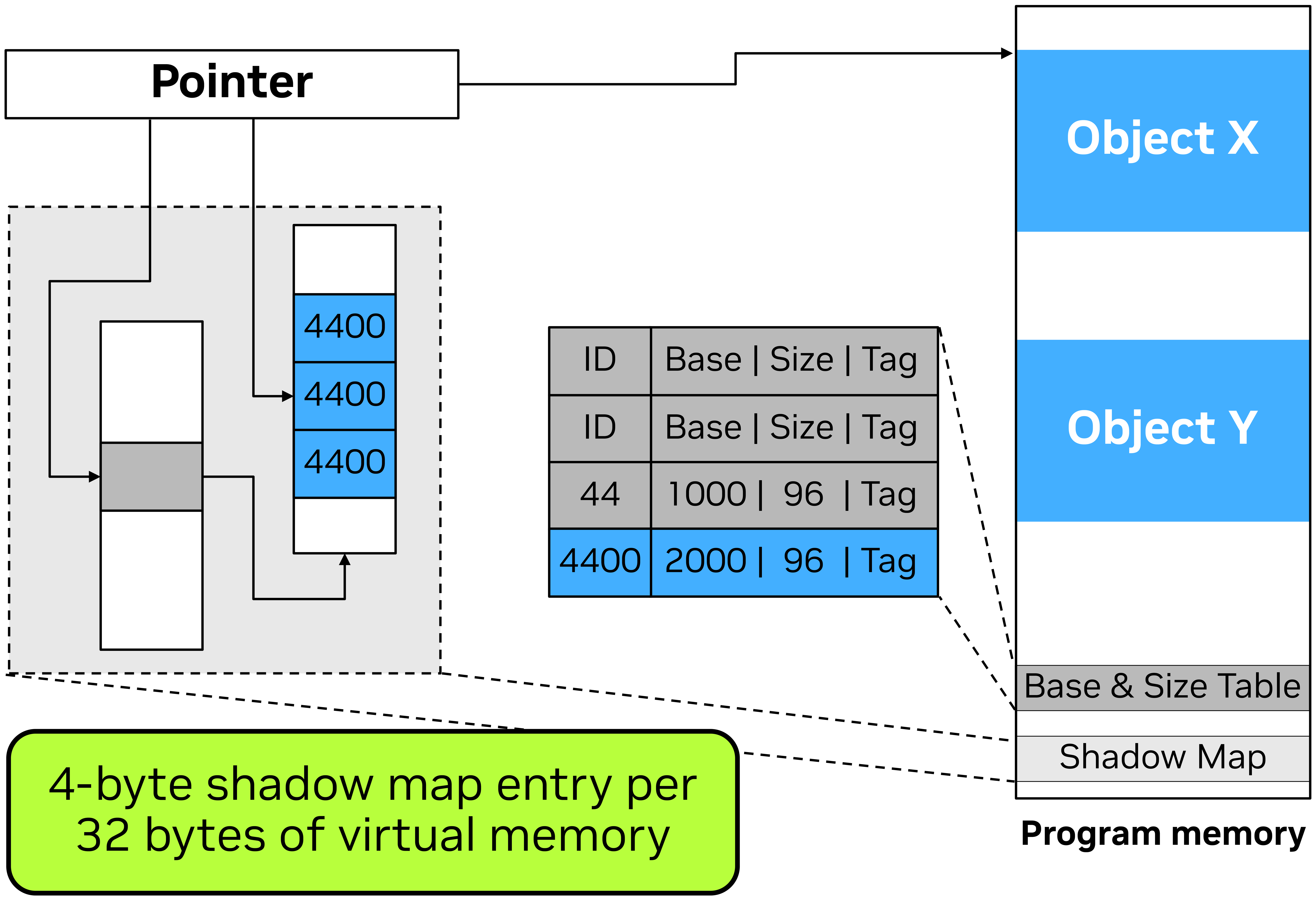



```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;

while(condition)
{

   t = *p;

   k = foo(t);

   p = p + k;

}

cudaFree(x);
.
.
.
p[0] = 0;
```

cuCatch uses a shadow map to establish a connection between an address and its BST entry

cuCatch Algorithm: Shadow Tagged Base & Bounds

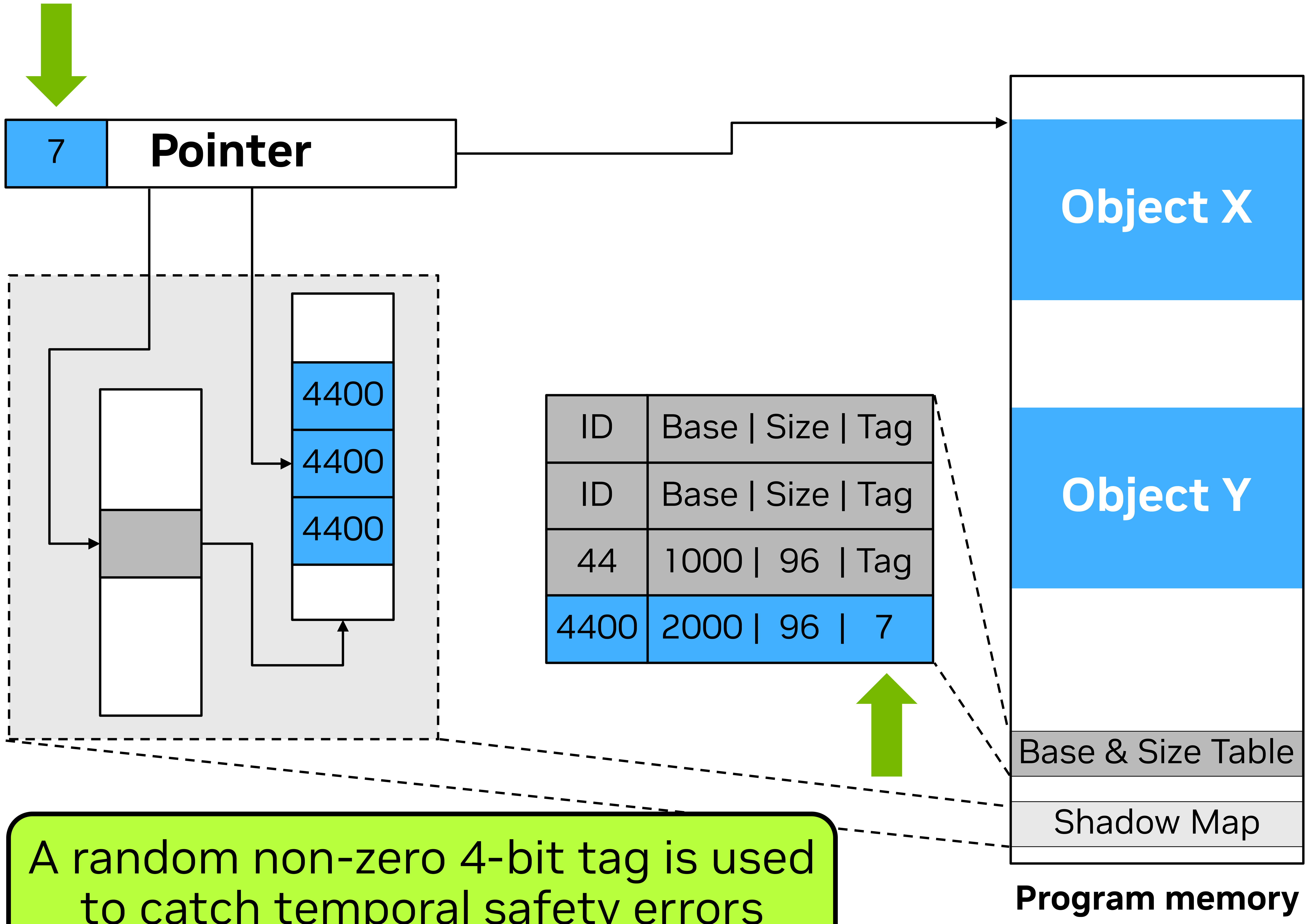
Memory Allocation: Runtime Support




```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;

while(condition)
{

   t = *p;

   k = foo(t);

   p = p + k;

}

cudaFree(x);
.
.
.
p[0] = 0;
```

cuCatch Algorithm: Shadow Tagged Base & Bounds

Memory Allocation: Runtime Support

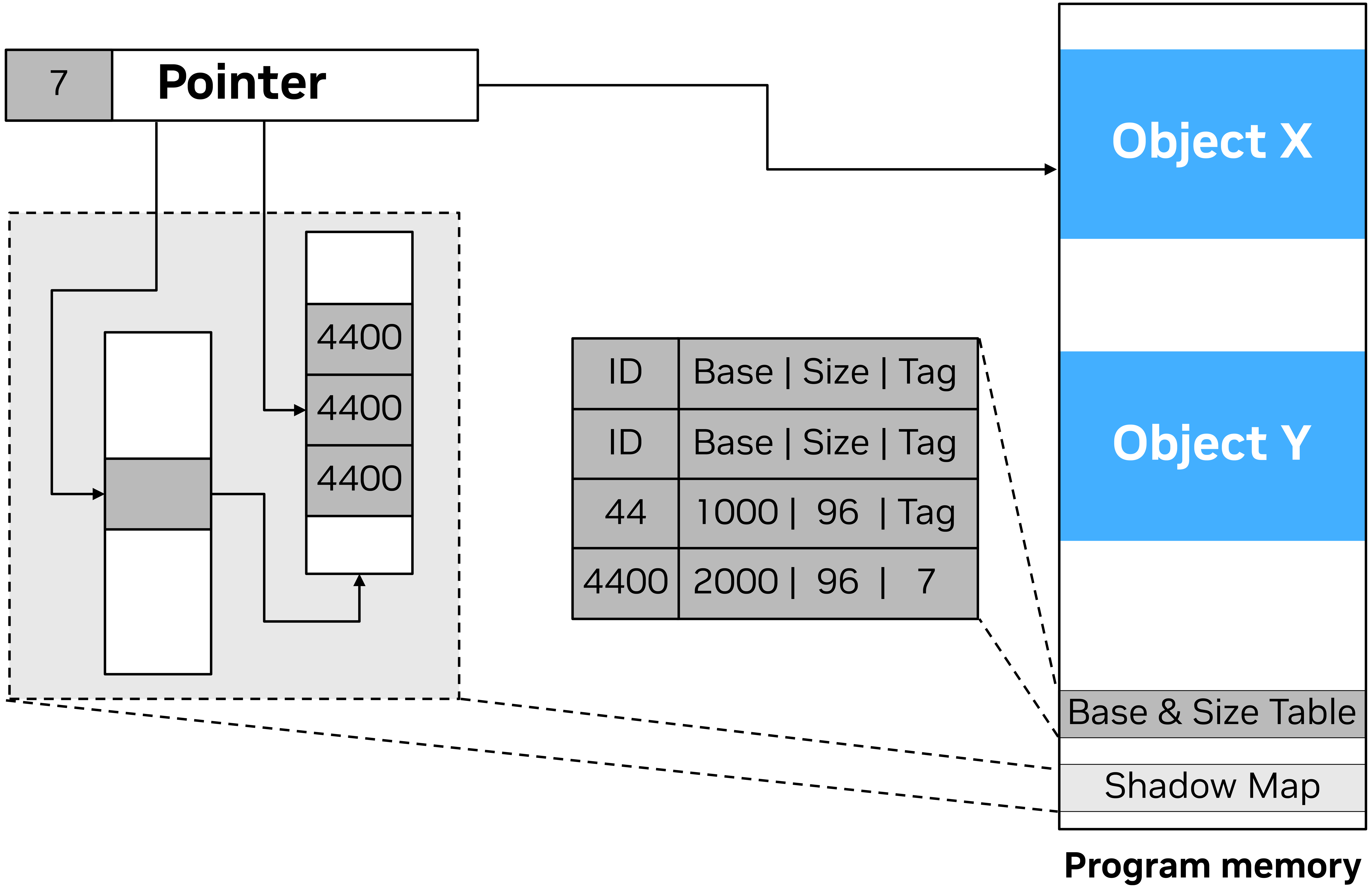



```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;

while(condition)
{
   t = *p;

   k = foo(t);

   p = p + k;

}

cudaFree(x);
.
.
.
p[0] = 0;
```

A random non-zero 4-bit tag is used to catch temporal safety errors

cuCatch Algorithm: Shadow Tagged Base & Bounds

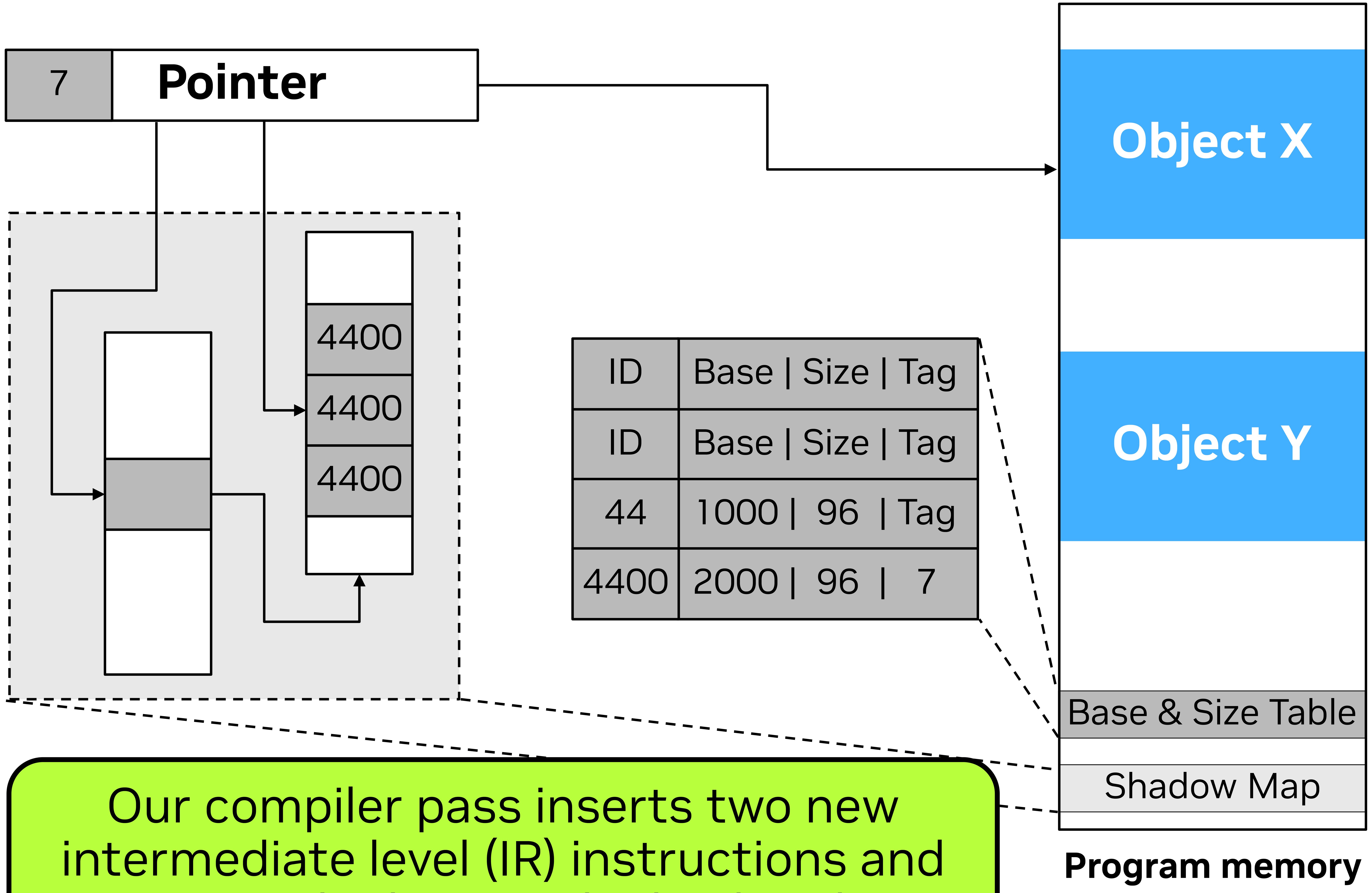
Memory Access




```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;

while(condition)
{

  t = *p;

  k = foo(t);

  p = p + k;

}

cudaFree(x);
.
.
.
p[0] = 0;
```

cuCatch Algorithm: Shadow Tagged Base & Bounds

Memory Access: Compiler Instrumentation



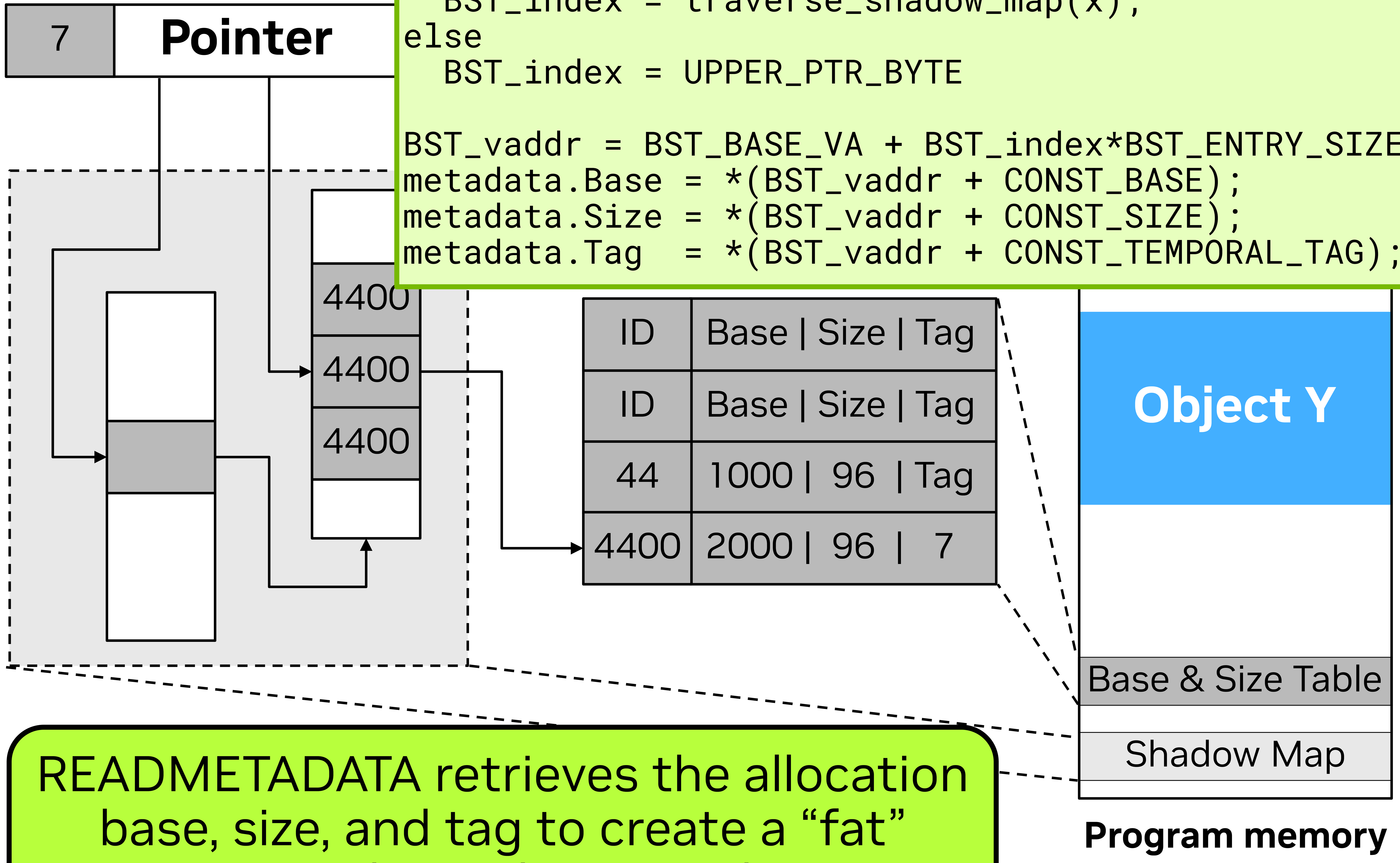
Our compiler pass inserts two new intermediate level (IR) instructions and expands them at the backend

```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;
READMETADATA metadata = [x]
while(condition)
{
  SAFETYCHECK metadata, [p]
  t = *p;

  k = foo(t);

  p = p + k;

}

cudaFree(x);
.
.
.
p[0] = 0;
```

cuCatch Algorithm: Shadow Tagged Base & Bounds

Memory Access: Compiler Instrumentation

```
UPPER_PTR_BYTE = [x] >> CONST_SHIFT;
if ( UPPER_PTR_BYTE >= 1 && UPPER_PTR_BYTE <= 15 )
  BST_index = traverse_shadow_map(x);
else
  BST_index = UPPER_PTR_BYTE

BST_vaddr = BST_BASE_VA + BST_index*BST_ENTRY_SIZE;
metadata.Base = *(BST_vaddr + CONST_BASE);
metadata.Size = *(BST_vaddr + CONST_SIZE);
metadata.Tag  = *(BST_vaddr + CONST_TEMPORAL_TAG);
```

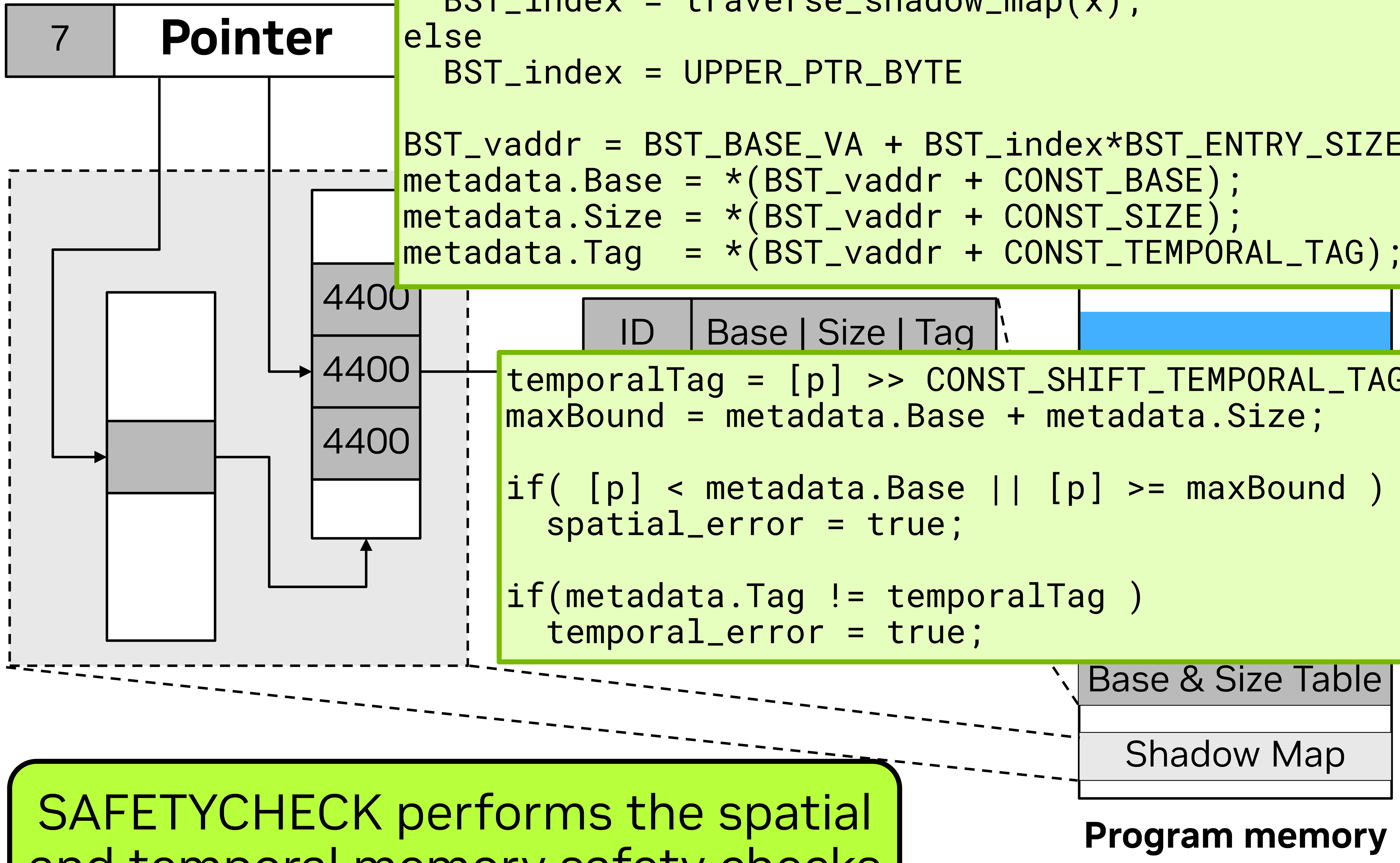



```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;
READMETADATA metadata = [x]
while(condition)
{
  SAFETYCHECK metadata, [p]
  t = *p;

  k = foo(t);

  p = p + k;

}

cudaFree(x);
.
.
.
p[0] = 0;
```

READMETADATA retrieves the allocation base, size, and tag to create a “fat” pointer without changing the ABI

cuCatch Algorithm: Shadow Tagged Base & Bounds

Memory Access: Compiler Instrumentation




```
UPPER_PTR_BYTE = [x] >> CONST_SHIFT;
if ( UPPER_PTR_BYTE >= 1 && UPPER_PTR_BYTE <= 15 )
  BST_index = traverse_shadow_map(x);
else
  BST_index = UPPER_PTR_BYTE

BST_vaddr = BST_BASE_VA + BST_index*BST_ENTRY_SIZE;
metadata.Base = *(BST_vaddr + CONST_BASE);
metadata.Size = *(BST_vaddr + CONST_SIZE);
metadata.Tag  = *(BST_vaddr + CONST_TEMPORAL_TAG);
```

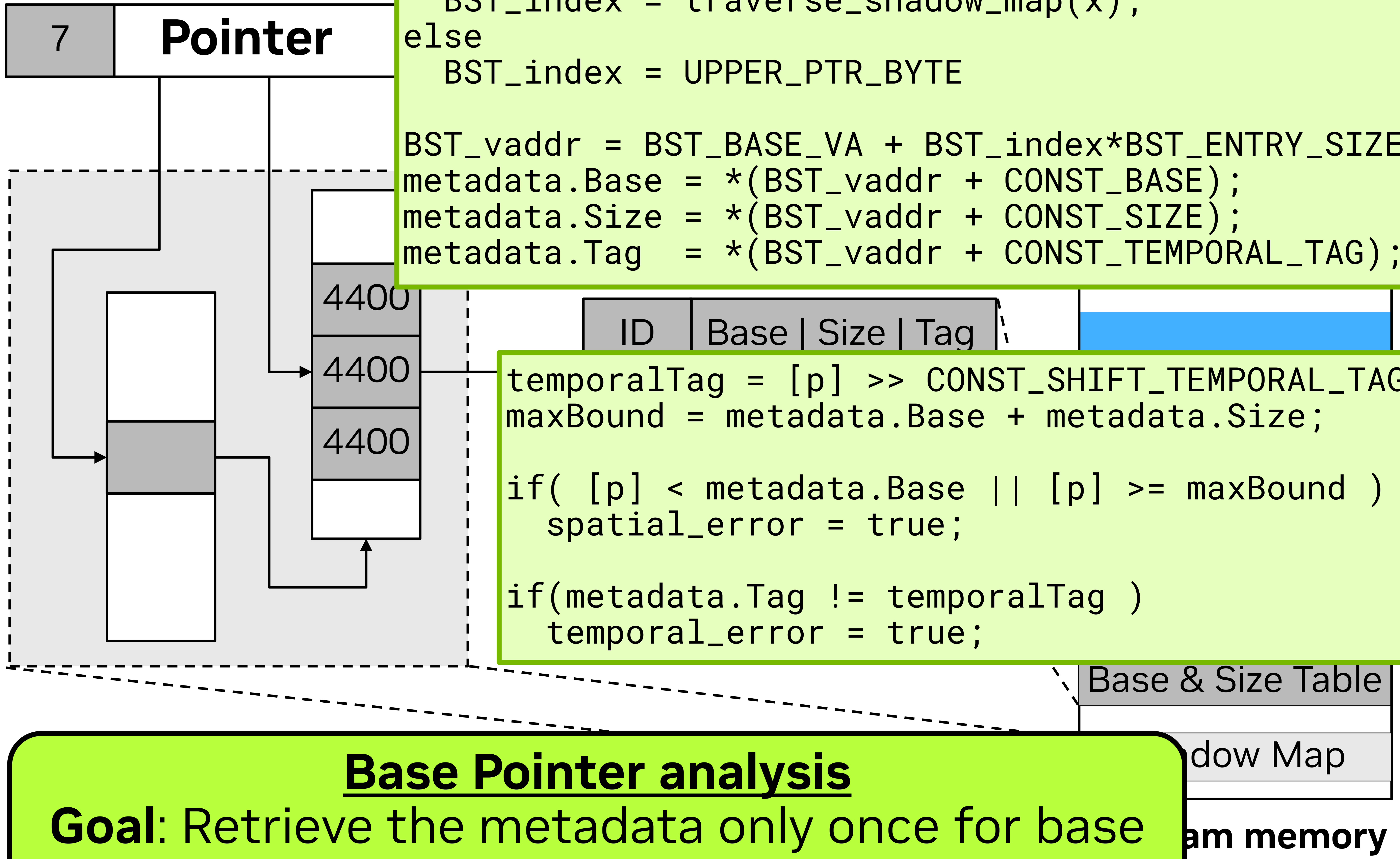
```
temporalTag = [p] >> CONST_SHIFT_TEMPORAL_TAG;
maxBound = metadata.Base + metadata.Size;

if( [p] < metadata.Base || [p] >= maxBound )
  spatial_error = true;

if(metadata.Tag != temporalTag )
  temporal_error = true;
```

```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;
READMETADATA metadata = [x]
while(condition)
{
  SAFETYCHECK metadata, [p]
  t = *p;

  k = foo(t);

  p = p + k;

}

cudaFree(x);
.
.
.
p[0] = 0;
```

SAFETYCHECK performs the spatial and temporal memory safety checks

cuCatch Algorithm: Shadow Tagged Base & Bounds

Memory Access: Compiler Instrumentation




```
UPPER_PTR_BYTE = [x] >> CONST_SHIFT;
if ( UPPER_PTR_BYTE >= 1 && UPPER_PTR_BYTE <= 15 )
  BST_index = traverse_shadow_map(x);
else
  BST_index = UPPER_PTR_BYTE

BST_vaddr = BST_BASE_VA + BST_index*BST_ENTRY_SIZE;
metadata.Base = *(BST_vaddr + CONST_BASE);
metadata.Size = *(BST_vaddr + CONST_SIZE);
metadata.Tag  = *(BST_vaddr + CONST_TEMPORAL_TAG);
```

```
temporalTag = [p] >> CONST_SHIFT_TEMPORAL_TAG;
maxBound = metadata.Base + metadata.Size;

if( [p] < metadata.Base || [p] >= maxBound )
  spatial_error = true;

if(metadata.Tag != temporalTag )
  temporal_error = true;
```

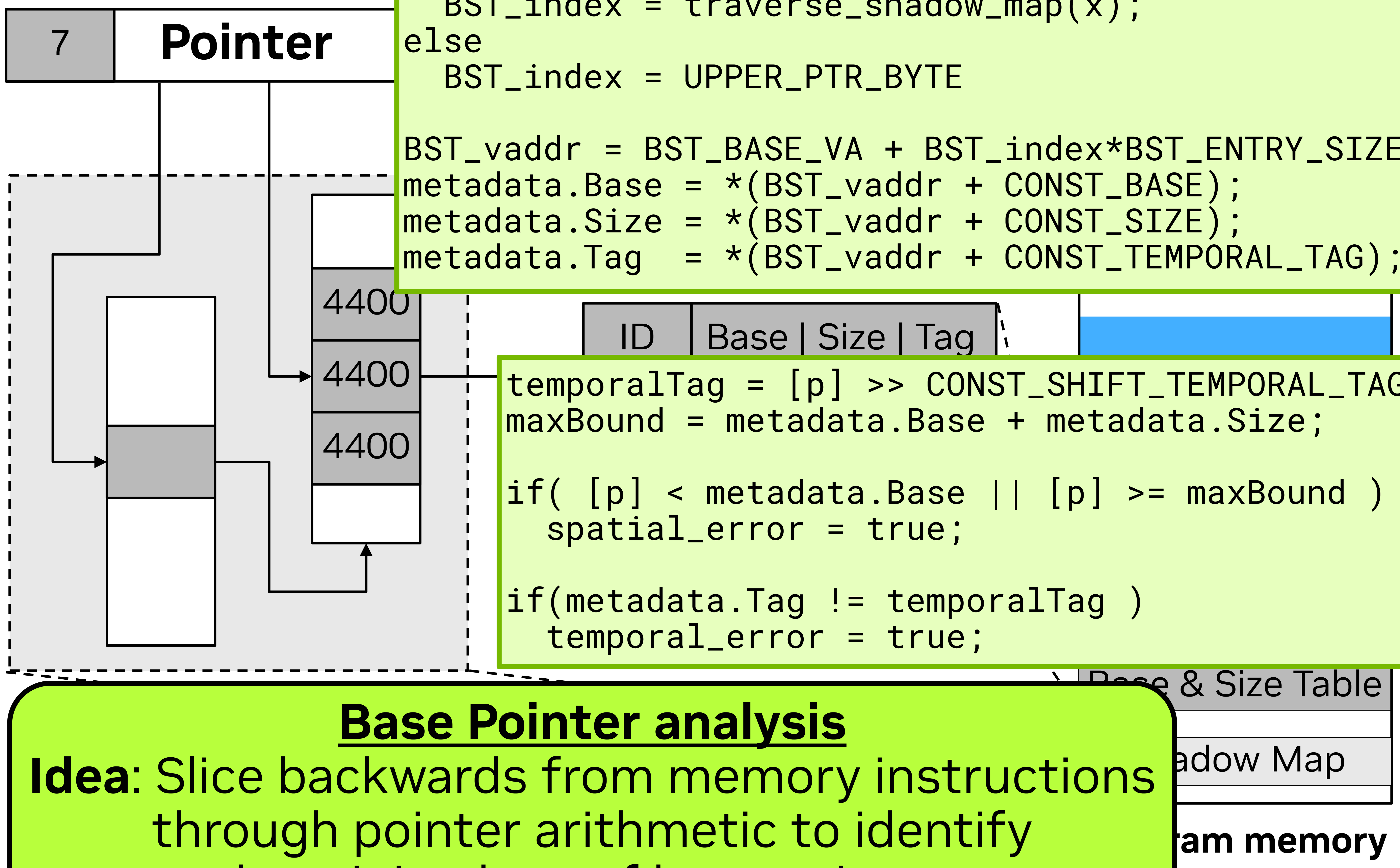
```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;
READMETADATA metadata = [x]
while(condition)
{
  SAFETYCHECK metadata, [p]
  t = *p;

  k = foo(t);

  p = p + k;

}

cudaFree(x);
.
.
.
p[0] = 0;
```

<u>Base Pointer analysis</u>

Goal: Retrieve the metadata only once for base pointers and propagate to ALL consumers

cuCatch Algorithm: Shadow Tagged Base & Bounds

Memory Access: Compiler Instrumentation




```
UPPER_PTR_BYTE = [x] >> CONST_SHIFT;
if ( UPPER_PTR_BYTE >= 1 && UPPER_PTR_BYTE <= 15 )
  BST_index = traverse_shadow_map(x);
else
  BST_index = UPPER_PTR_BYTE

BST_vaddr = BST_BASE_VA + BST_index*BST_ENTRY_SIZE;
metadata.Base = *(BST_vaddr + CONST_BASE);
metadata.Size = *(BST_vaddr + CONST_SIZE);
metadata.Tag  = *(BST_vaddr + CONST_TEMPORAL_TAG);
```

```
temporalTag = [p] >> CONST_SHIFT_TEMPORAL_TAG;
maxBound = metadata.Base + metadata.Size;

if( [p] < metadata.Base || [p] >= maxBound )
  spatial_error = true;

if(metadata.Tag != temporalTag )
  temporal_error = true;
```

```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;
READMETADATA metadata = [x]
while(condition)
{
  SAFETYCHECK metadata, [p]
  t = *p;

  k = foo(t);

  p = p + k;

}

cudaFree(x);
.
.
.
p[0] = 0;
```

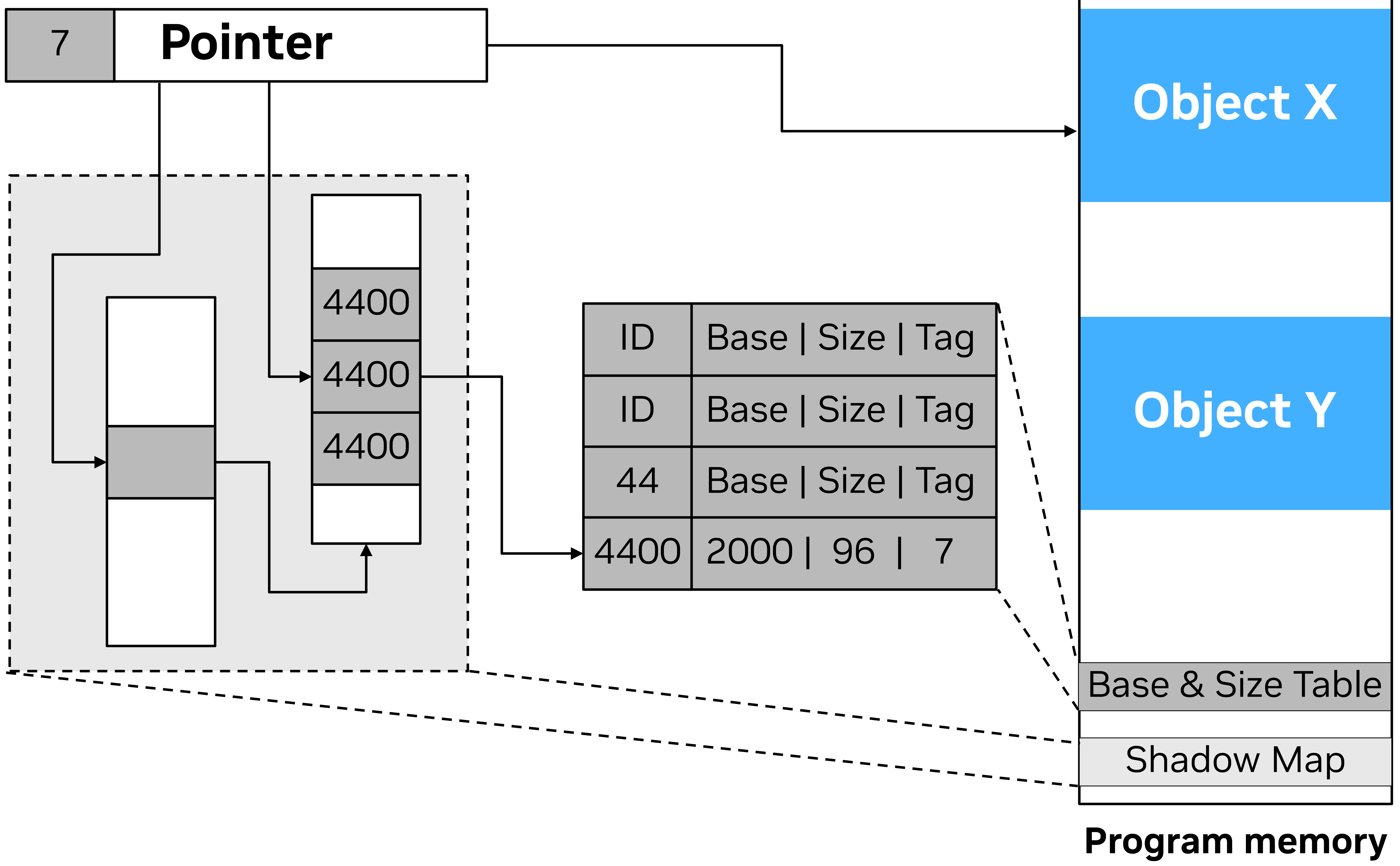
Base Pointer analysis

Idea: Slice backwards from memory instructions through pointer arithmetic to identify the minimal set of base pointers

cuCatch Algorithm: Shadow Tagged Base & Bounds

Memory Access: Compiler Instrumentation

Retrieving the metadata as early as possible offers **performance** and **security** benefits



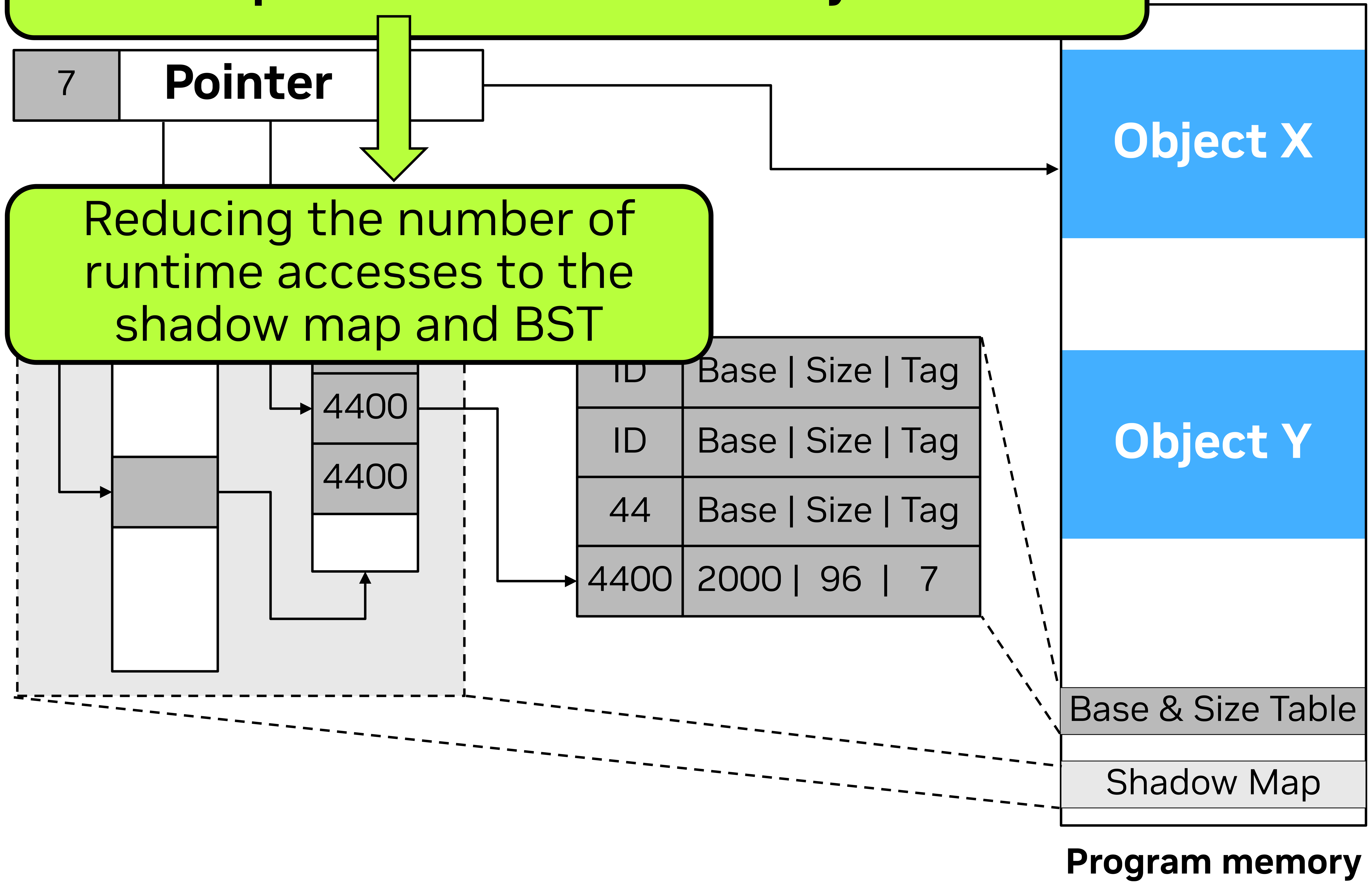

```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;
READMETADATA metadata = [x]
while(condition)
{
  SAFETYCHECK metadata, [p]
  t = *p;

  k = foo(t);

  p = p + k;

}

cudaFree(x);
.
.
.
p[0] = 0;
```

cuCatch Algorithm: Shadow Tagged Base & Bounds

Memory Access: Compiler Instrumentation

Retrieving the metadata as early as possible offers **performance** and **security** benefits

Reducing the number of runtime accesses to the shadow map and BST




```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;
READMETADATA metadata = [x]
while(condition)
{
  SAFETYCHECK metadata, [p]
  t = *p;

  k = foo(t);

  p = p + k;

}

cudaFree(x);
.
.
.
p[0] = 0;
```

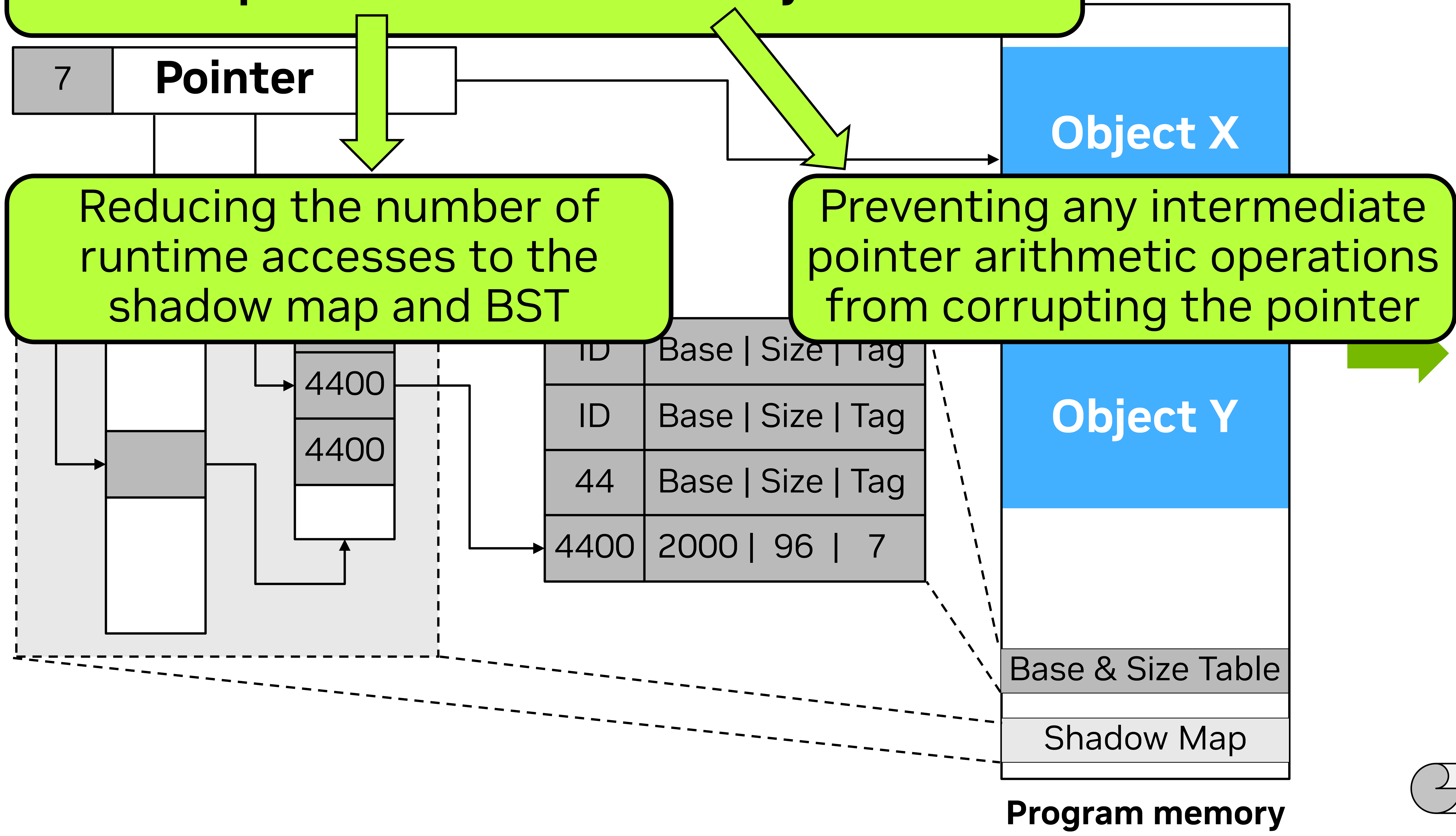
cuCatch Algorithm: Shadow Tagged Base & Bounds

Memory Access: Compiler Instrumentation

Retrieving the metadata as early as possible offers **performance** and **security** benefits

Reducing the number of runtime accesses to the shadow map and BST

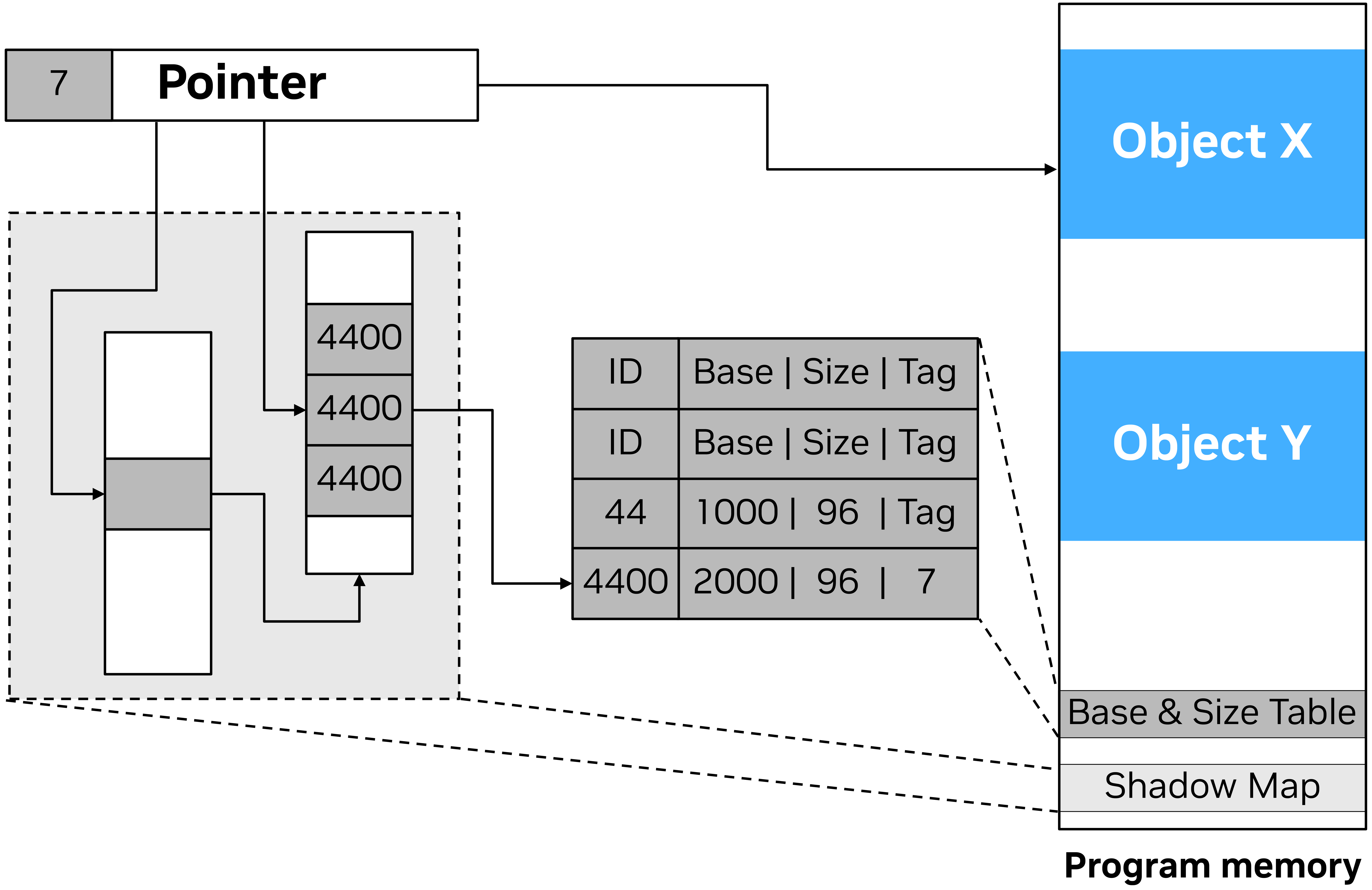
Preventing any intermediate pointer arithmetic operations from corrupting the pointer




```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;
READMETADATA metadata = [x]
while(condition)
{
  SAFETYCHECK metadata, [p]
  t = *p;

  k = foo(t);

  p = p + k;

}

cudaFree(x);
.
.
.
p[0] = 0;
```

cuCatch Algorithm: Shadow Tagged Base & Bounds

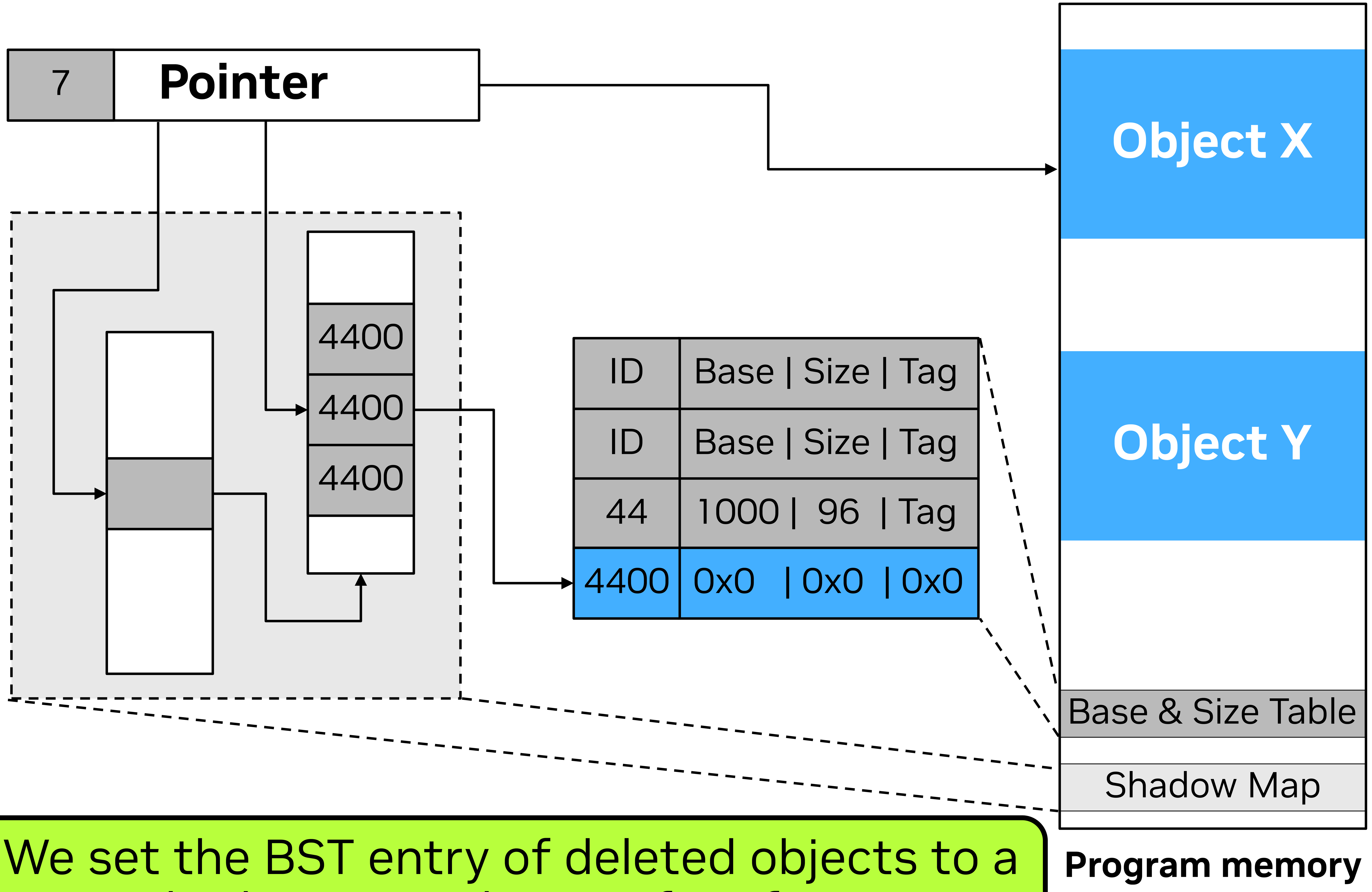
Memory Deallocation: Runtime Support




```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;
READMETADATA metadata = [x]
while(condition)
{
  SAFETYCHECK metadata, [p]
  t = *p;

  k = foo(t);

  p = p + k;

}

cudaFree(x);
.
.
.
p[0] = 0;
```

cuCatch Algorithm: Shadow Tagged Base & Bounds

Memory Deallocation: Runtime Support

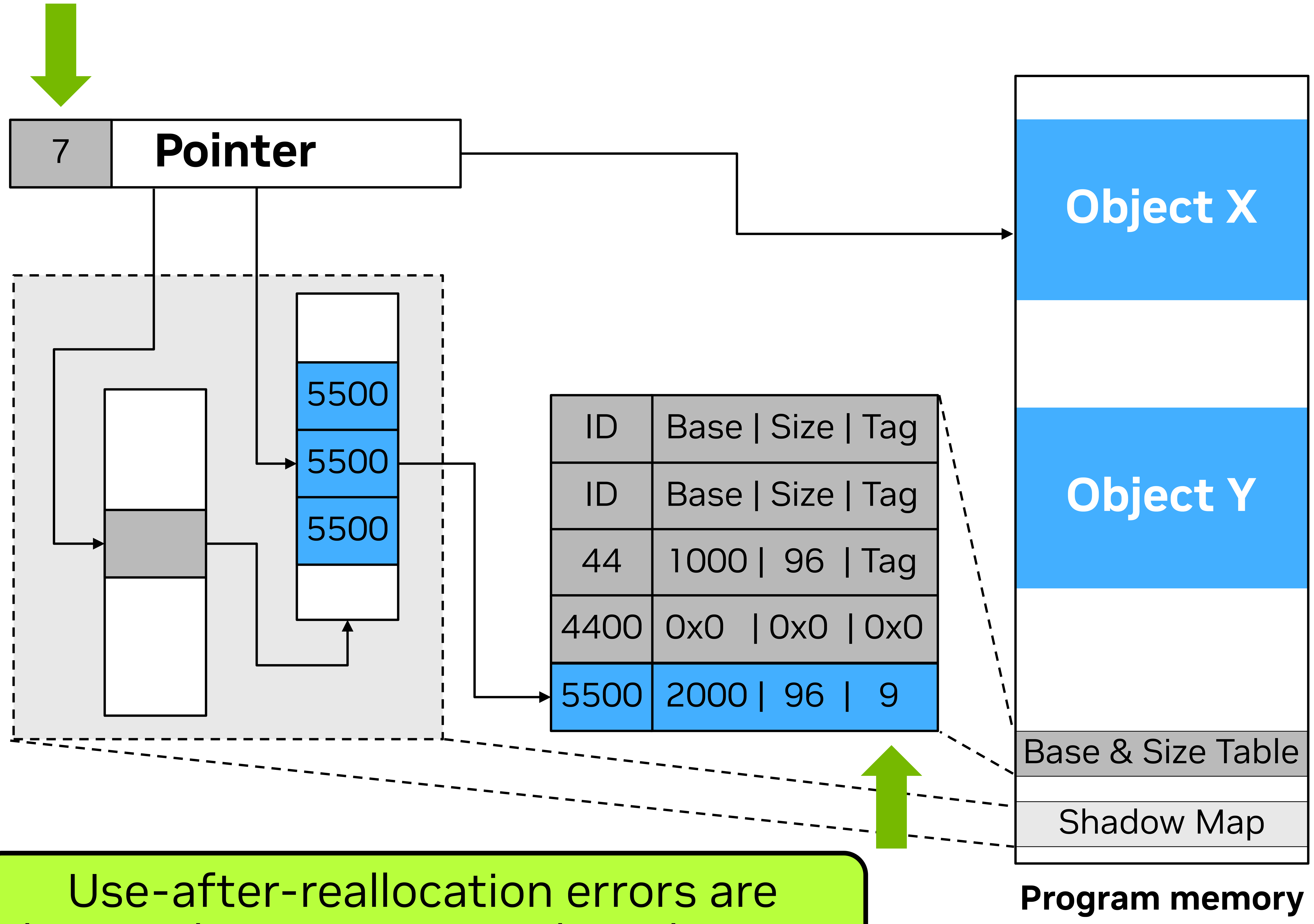


We set the BST entry of deleted objects to a special value to catch use-after-free errors

```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;
READMETADATA metadata = [x]
while(condition)
{
  SAFETYCHECK metadata, [p]
  t = *p;

  k = foo(t);

  p = p + k;

}

cudaFree(x);
.
.
.
p[0] = 0;
```

cuCatch Algorithm: Shadow Tagged Base & Bounds

Memory Deallocation: Runtime Support

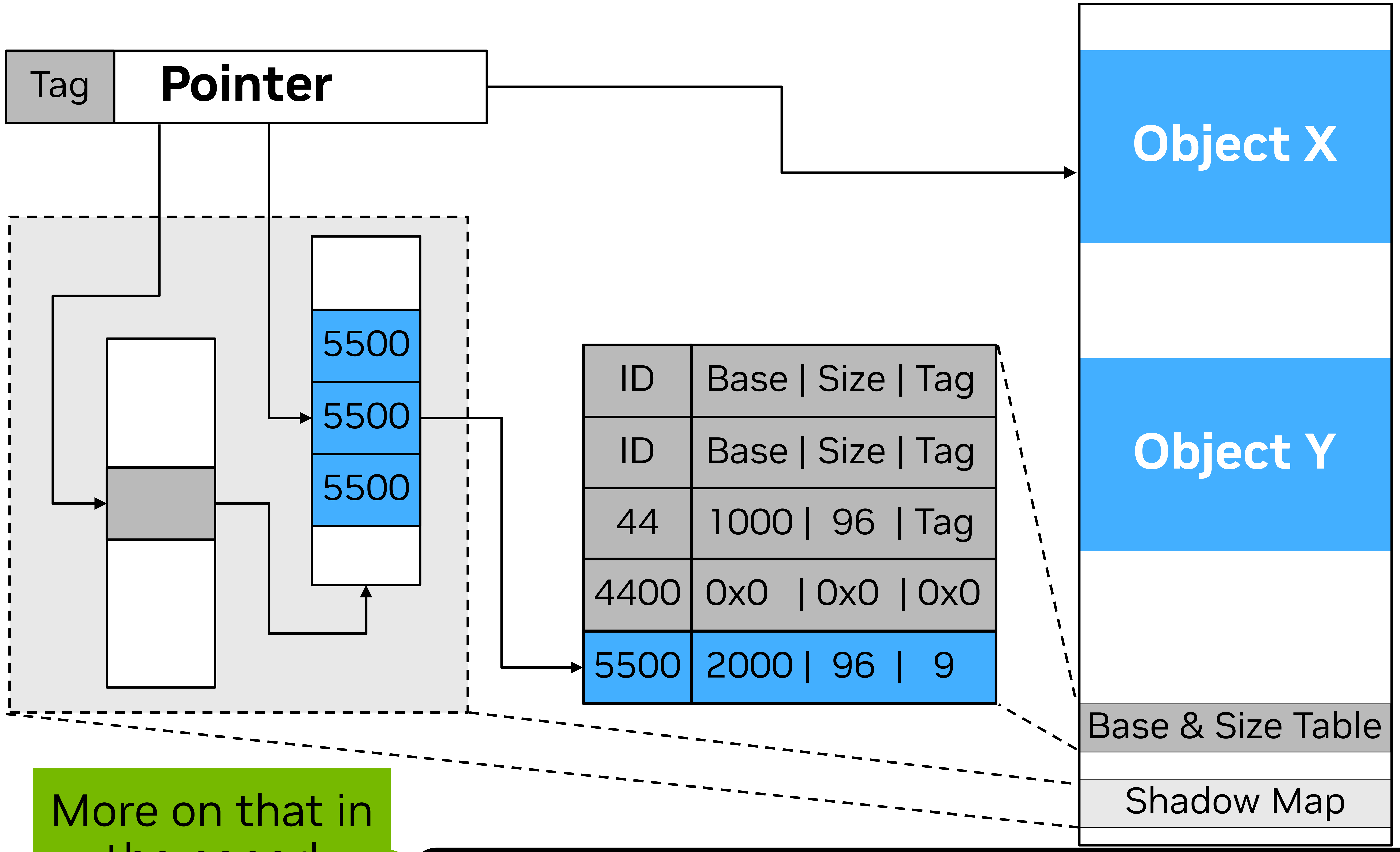



```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;
READMETADATA metadata = [x]
while(condition)
{
  SAFETYCHECK metadata, [p]
  t = *p;

  k = foo(t);

  p = p + k;

}

cudaFree(x);
.
.
.
p[0] = 0;
```

Use-after-reallocation errors are detected via comparing the 4-bit Tags

cuCatch Algorithm: Shadow Tagged Base & Bounds

Optimizations




```
int *x;
cudaMalloc(&x, N * sizeof(int));
.
.
.
p = x;
READMETADATA metadata = [x]
while(condition)
{
  SAFETYCHECK metadata, [p]
  t = *p;

  k = foo(t);

  p = p + k;

}

cudaFree(x);
.
.
.
p[0] = 0;
```

More on that in the paper!

cuCatch handles global, shared, and local GPU allocations & performs multiple compile-time optimizations

cuCatch Implementation & Evaluation

cuCatch Implementation & Evaluation

Error Detection Coverage: How many memory safety errors can cuCatch detect?

Performance: What is the runtime cost of cuCatch? and how does it compare to other tools?

Evaluation

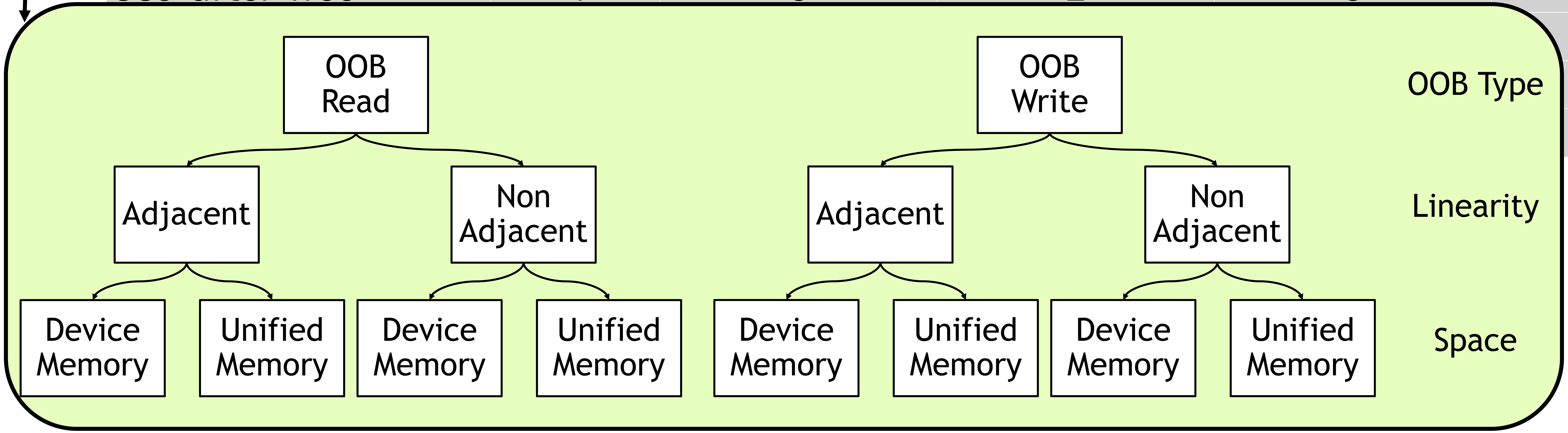
- cuCatch Overview
- **Error Detection Coverage**
- Evaluation Results
- Conclusion

Error Detection Coverage

Benchmark type	Total tests	Number of detected tests per tool			
		Baseline	Compute Sanitizer	GPUShield [ISCA 2022]	cuCatch
Global memory OOB	8	0	4	8	8
Local memory OOB	16	0	4	12	12
Shared memory OOB	12	0	4	0	10
Intra-allocation OOB	8	0	0	0	0
Use-after-free	4	0	2	0	2
Use-after-scope	4	0	2	0	4
Invalid free	2	2	2	2	2
Double free	2	2	2	2	2
Detection rate	56	7.1%	35.7%	42.8%	**71.4%**

Error Detection Coverage

Benchmark type	Total tests	Number of detected tests per tool			
		Baseline	Compute Sanitizer	GPUShield [ISCA 2022]	cuCatch
Global memory OOB	8	0	4	8	8
Local memory OOB	16	0	4	12	12
Shared memory OOB	12	0	4	0	10
Intra-allocation OOB	8	0	0	0	0
Use-after-free	4	0	2	0	2
					4
					2
					2
					71.4%



Error Detection Coverage

Benchmark type	Total tests	Number of detected tests per tool			
		Baseline	Compute Sanitizer	GPUShield [ISCA 2022]	cuCatch
Global memory OOB	8	0	4	8	8
Local memory OOB	16	0	4	12	12
Shared memory OOB	12	0	4	0	10
Intra-allocation OOB	8	0	0	0	0
Use-after-free	4	0	2	0	2
Use-after-scope	4	0	2	0	4
Invalid free	2	2	2	2	2
Double free	2	2	2	2	2
Detection rate	56	7.1%	35.7%	42.8%	**71.4%**

cuCatch offers the highest error detection coverage

Error Detection Coverage

Benchmark type	Total tests	Number of detected tests per tool			
		Baseline	Compute Sanitizer	GPUShield [ISCA 2022]	cuCatch
Global memory OOB	8	0	4	8	8
Local memory OOB	16	0	4	12	12
Shared memory OOB	12	0	4	0	10
Intra-allocation OOB	8	0	0	0	0
Use-after-free	4	0	2	0	2
Use-after-scope	4	0	2	0	4
Invalid free	2	2	2	2	2
Double free	2	2	2	2	2
Detection rate	56	7.1%	35.7%	42.8%	**71.4%**

Limitations: intra-allocation OOB & overflows in *dynamically* allocated buffers in shared memory

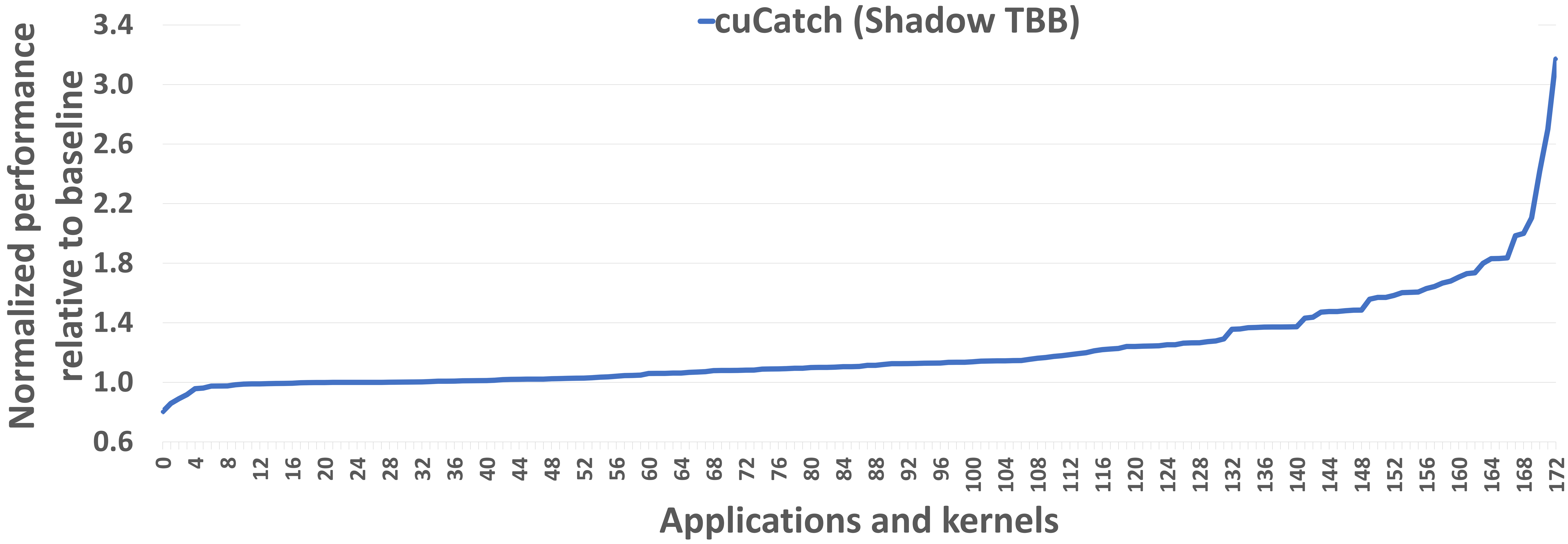
- cuCatch Overview
- Error Detection Coverage
- **Evaluation Results**
- Conclusion

Experimental Setup

- **Benchmarks**
  - CUDA kernels from various workload segments
    - Scientific computing: namd, amber18, AMG, FUN3D, Laghos, lammps, Relion
    - Commercial: 5G decoding
    - Visualization: Optix
  - The PolyBench-ACC suite.
  - Most of the CUDA-samples SDK.
- **Platform**
  - NVIDIA GeForce RTX 2080Ti GPU (1710 MHz core clock)

cuCatch: Performance Evaluation

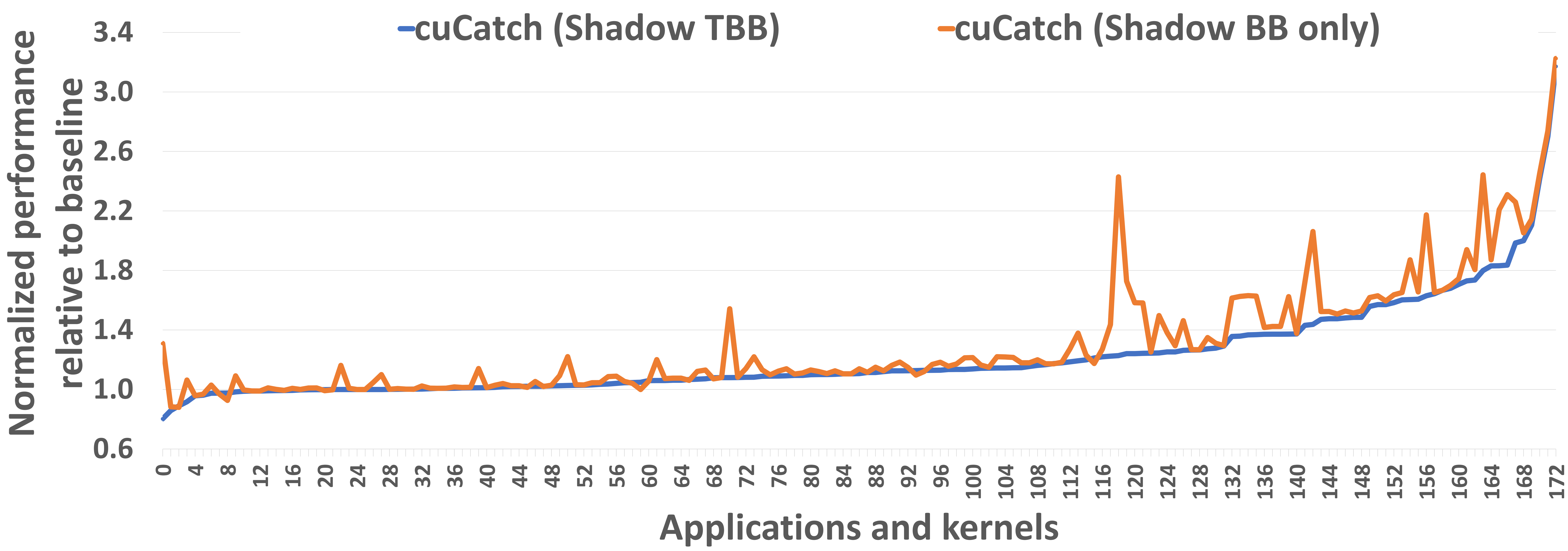
Runtime Overheads



cuCatch (with Shadow TBB) introduces 19% slowdowns on average

cuCatch: Performance Evaluation

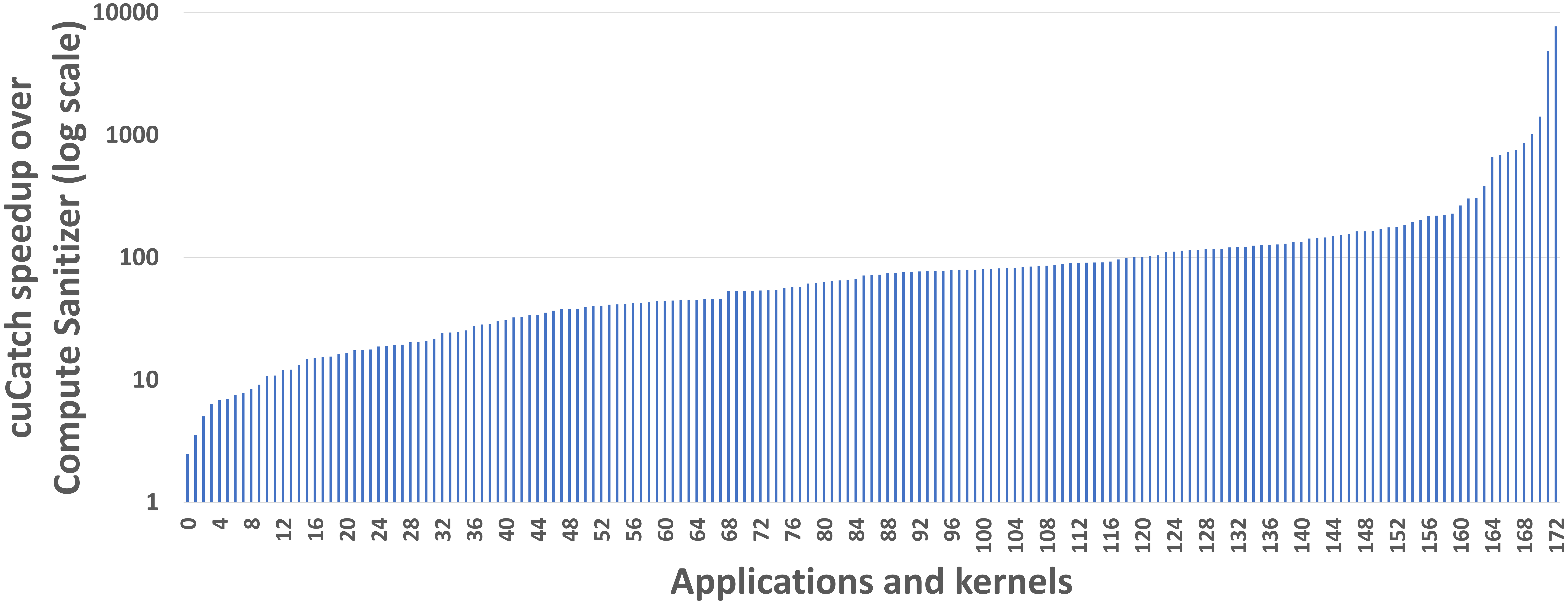
Sensitivity Analysis



cuCatch (with Shadow BB only) introduces 25% slowdowns on average

cuCatch: Performance Evaluation

Comparison With the State-of-the-art Error Detection Tools



cuCatch is orders of magnitude faster than Compute Sanitizer's memcheck tool

- cuCatch Overview
- Error Detection Coverage
- Evaluation Results
- **Conclusion**

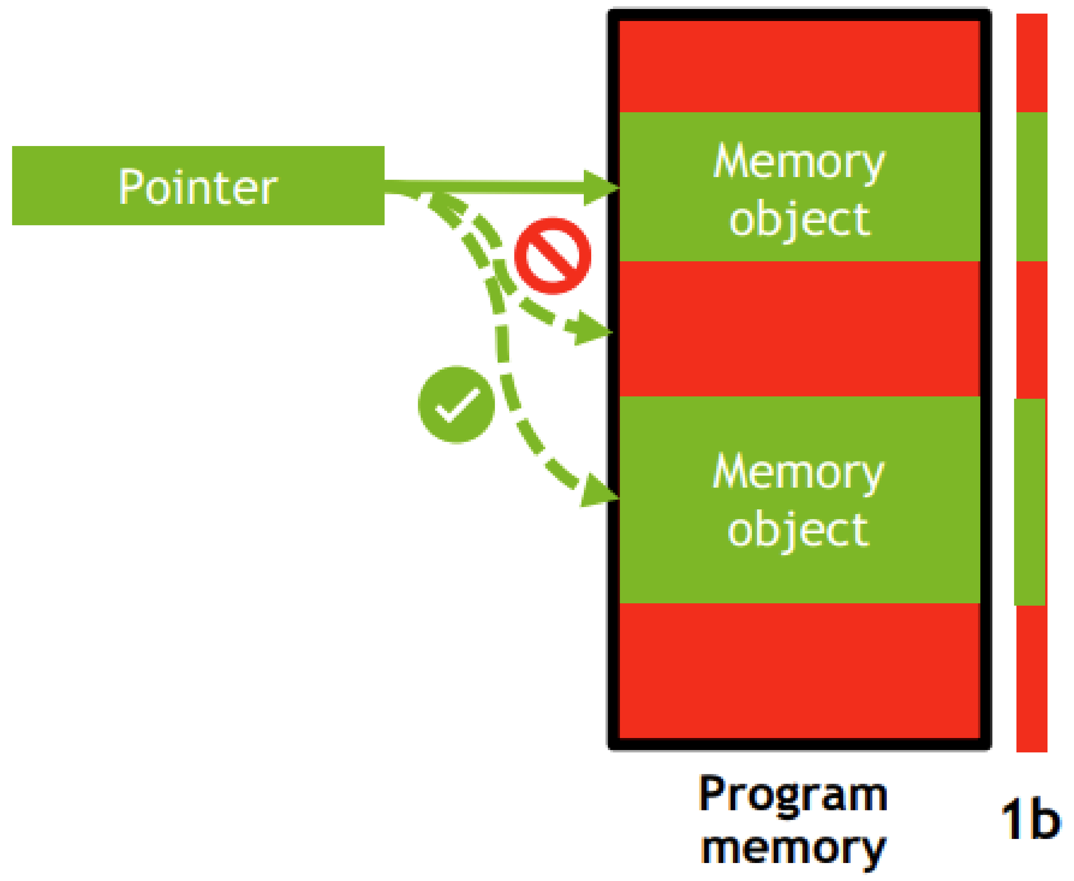
Conclusion

- **cuCatch** combines a novel memory safety algorithm with an efficient compiler-based instrumentation to provide GPU users with a debugging tool:
  - **Has low runtime overheads**
  - **Provides high error detection coverage**
  - **Scales to arbitrary number of allocations**
  - **Requires no software/hardware changes**

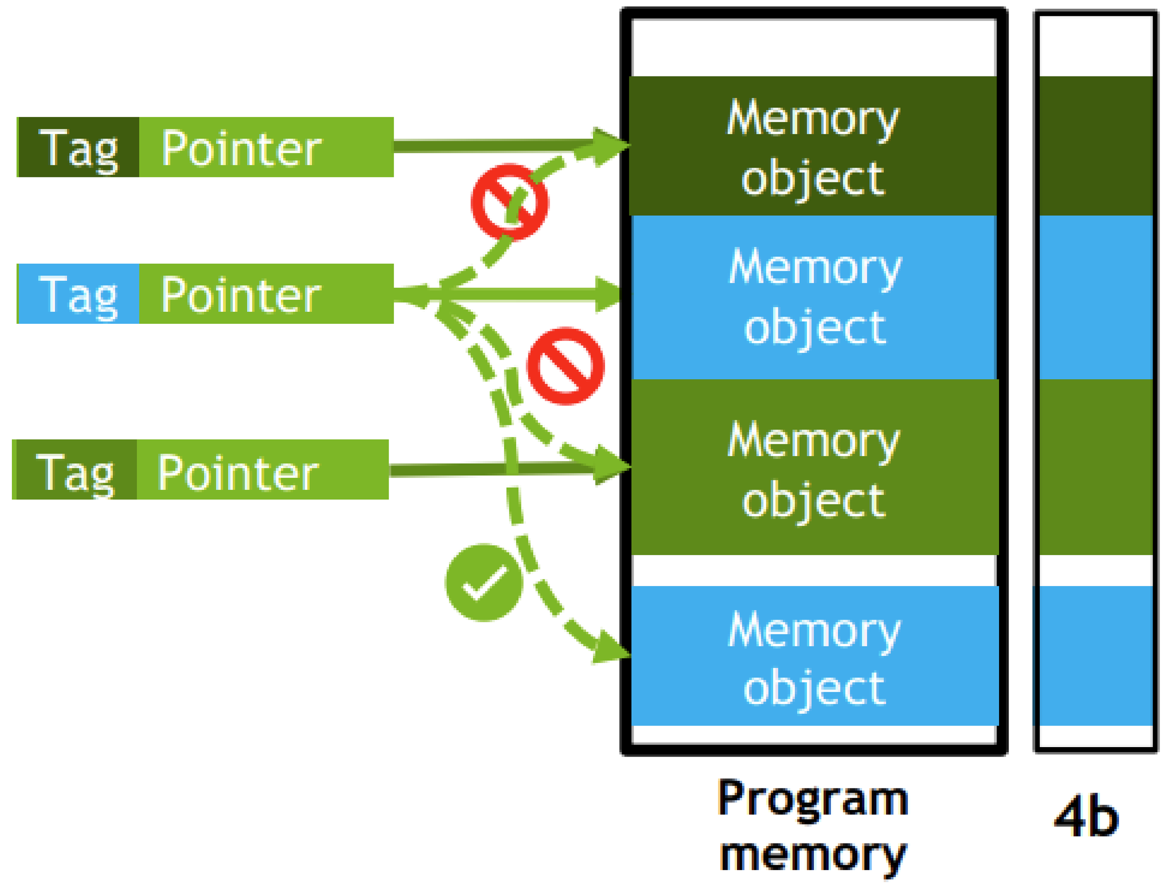
NVIDIA®

BACKUP SLIDES

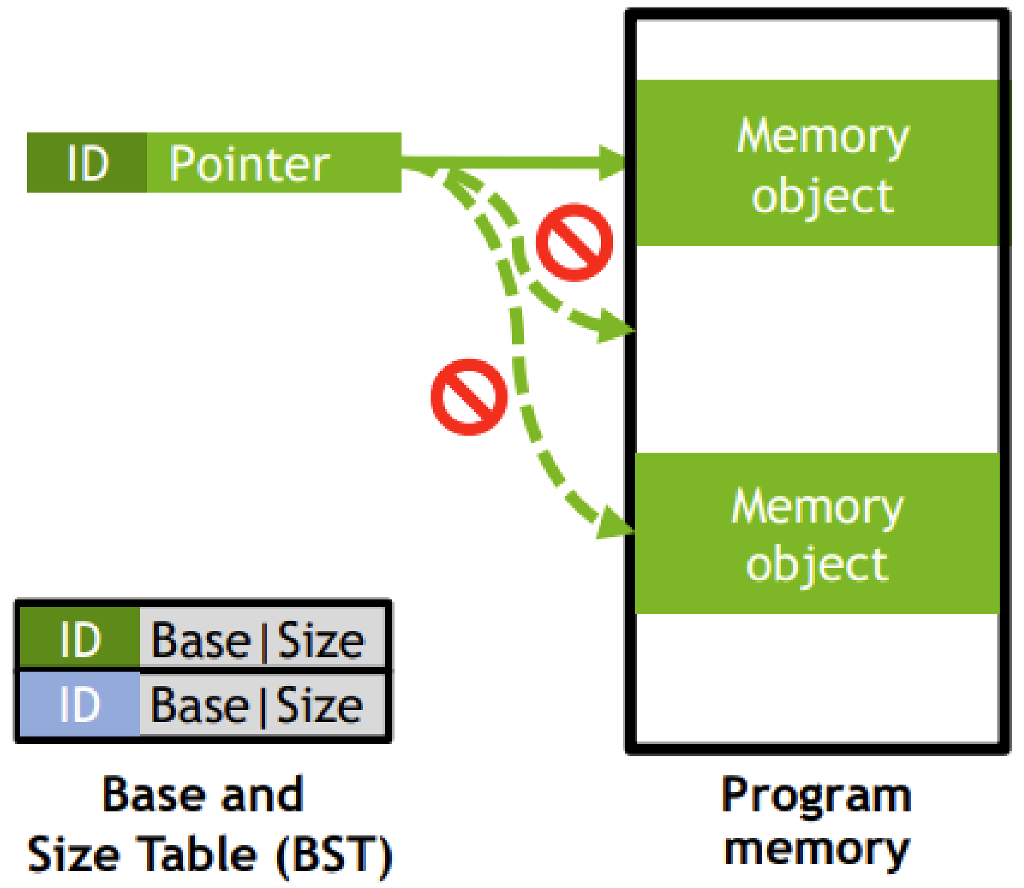
Memory Safety Algorithms



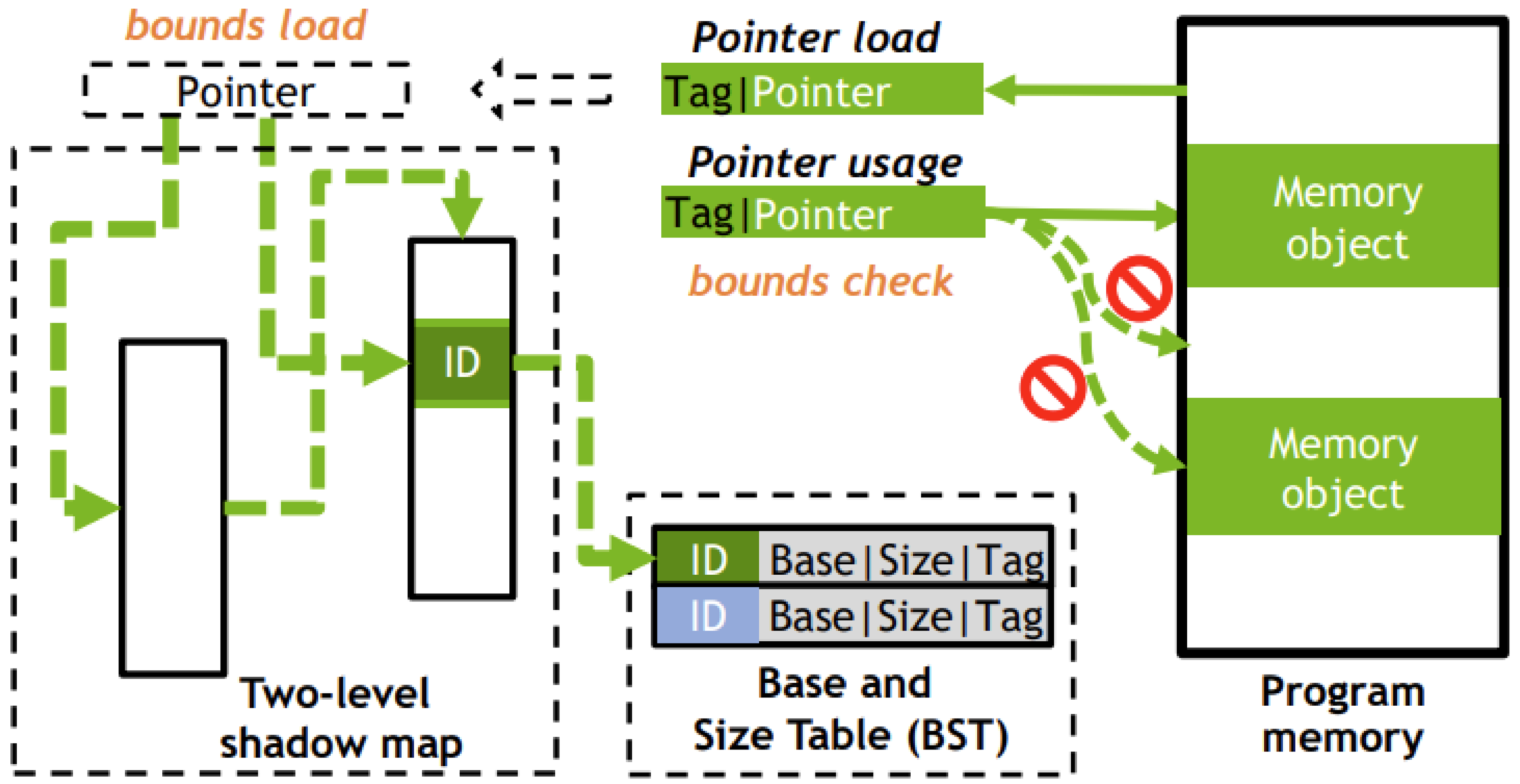
(a) Tripwires.



(b) Memory tagging.



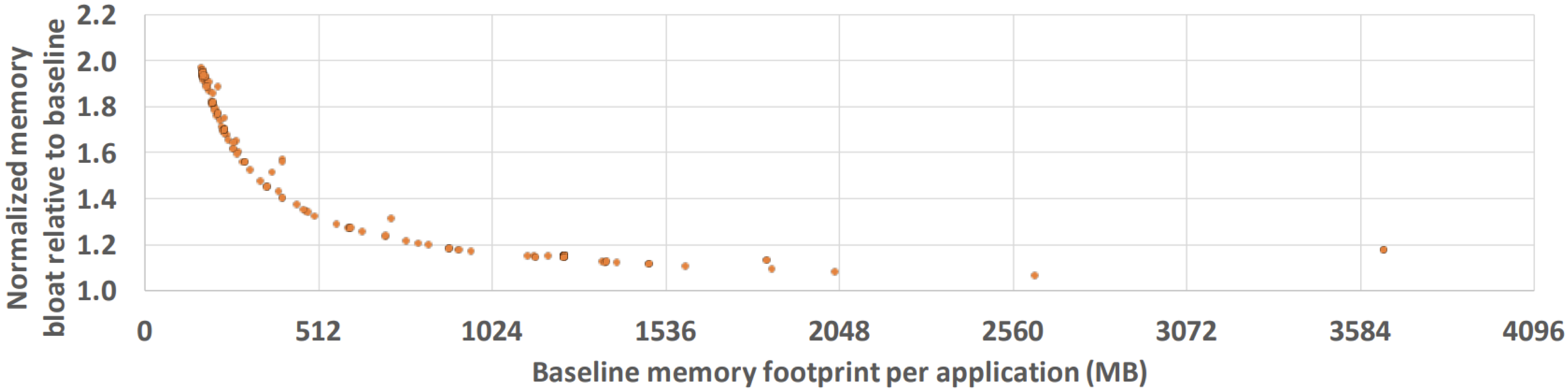
(c) Tagged base & bounds.



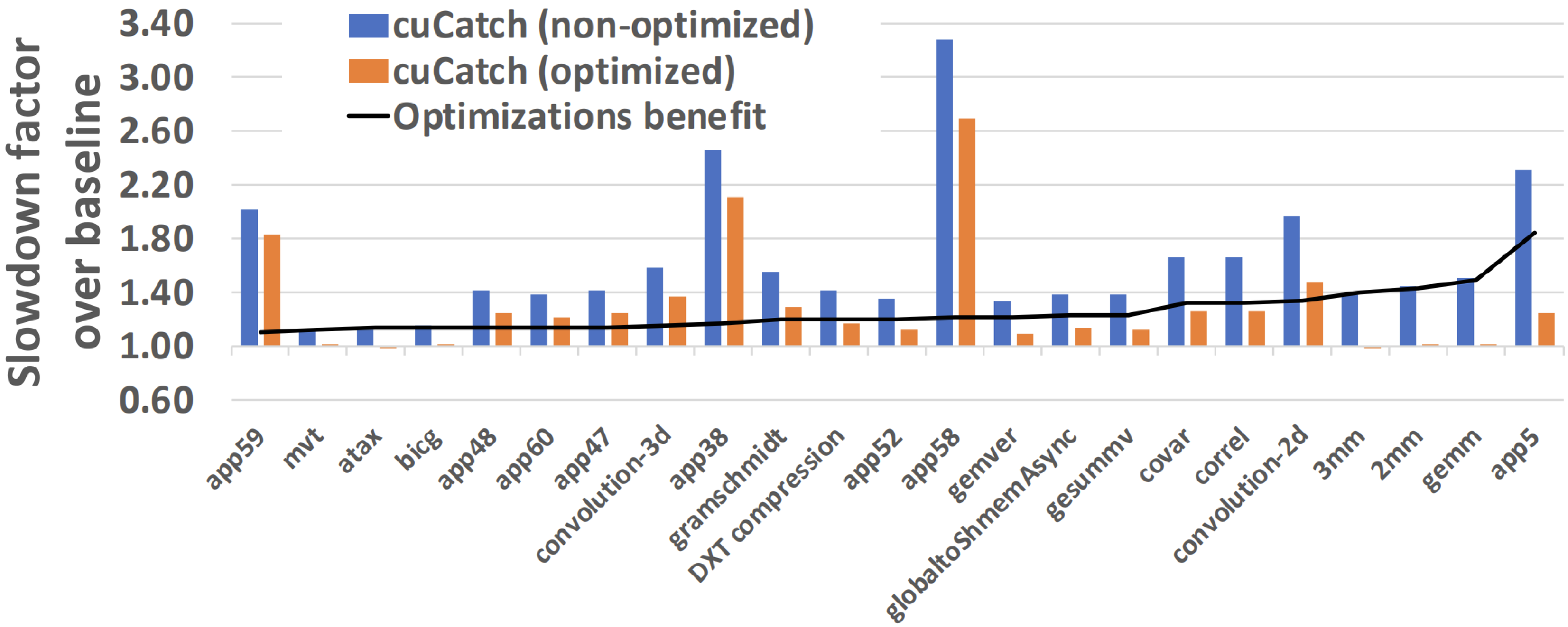
(d) Shadow tagged base & bounds.

cuCatch: Memory Evaluation

cuCatch Memory Overheads Normalized to Baseline Execution



cuCatch: Optimizations Evaluation



Related Work

Proposal	Platform	Instrumentation Level †	Spatial Safety *	Temporal Safety §	Metadata Requirements	Memory Overhead	Performance Overhead
REST	CPU	Hardware	◒	◒	8–64B token per object	∝ blocklisted memory	∝ # of (dis)arm insns.
Califorms	CPU	ISA	◒	◒	1-7B per field	∝ blocklisted memory	∝ # of BLOC insns.
ARM MTE	CPU	ISA	◒	◒	4 bits per 16B region	∝ prog. mem. footprint	∝ # of tag (un)set ops
CHERI	CPU	ISA	●	○	Ptr size is 2-4X	∝ # of ptrs	∝ # of ptr ops
CHERIvoke	CPU	ISA	○	●	Ptr size is 2-4X	∝ # of ptrs	∝ # of ptr ops
Intel MPX	CPU	ISA	●	○	2 words per ptr	∝ # of ptrs	∝ # of ptr derefs
CHEx86	CPU	Hardware	●	◒	2 words per ptr	∝ # of objects & ptrs	∝ # of ptr derefs
No-FAT	CPU	ISA	●	◒	1KB per process sizes table	∝ padding objects to the nearest size	∝ # of ptr derefs
AOS	CPU	ISA	●	◒	8B bounds per ptr	∝ # of ptrs	∝ # of ptr derefs
Valgrind	CPU	Binary	◒	○	1B per 8B region	∝ prog. mem. footprint	∝ # of ptr derefs
SoftBound	CPU	Compiler	●	○	2 words per ptr	∝ # of ptrs	∝ # of ptr derefs
Address Sanitizer	CPU	Compiler	◒	○	1B per 8B region	∝ prog. mem. footprint	∝ # of ptr derefs
GPU Shield	GPU	Hardware	●	○	2 words per object	∝ # of objects	∝ # of ptr derefs
Compute Sanitizer	GPU	Binary	◒	○	2 words per object	∝ # of objects	∝ # of ptr derefs
GMOD	GPU	Compiler	◒	○	8B canary per object	∝ blocklisted memory	∝ # of ptr derefs
clARMOR	GPU	Compiler	◒	○	8B canary per object	∝ blocklisted memory	∝ # of ptr derefs
cuCatch	GPU	Compiler	●	◒	32 bits per 32B region	∝ prog. mem. footprint	∝ # of ptr derefs

† Hardware - hardware-only changes; Compiler - compiler-level changes; Binary - DBI; ISA - hardware and compiler changes.
* ● - Complete (Linear and non-linear overflows); ◒ - Linear only; ○ - No coverage.
§ ● - Complete; ◒ - Partial coverage; ○ - No coverage.