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

MOHAMED TAREK IBN ZIAD, NVIDIA, USA
SANA DAMANI, NVIDIA, USA
AAMER JALEEL, NVIDIA, USA
STEPHEN W. KECKLER, NVIDIA, USA
MARK STEPHENSON, NVIDIA, USA

CUDA, OpenCL, and OpenACC are the primary means of writing general-purpose software for NVIDIA GPUs, all of which are subject to the same well-documented memory safety vulnerabilities currently plaguing software written in C and C++. One can argue that the GPU execution environment makes software development more error prone. Unlike C and C++, CUDA features multiple, distinct memory spaces to map to the GPU's unique memory hierarchy, and a typical CUDA program has thousands of concurrently executing threads. Furthermore, the CUDA platform has fewer guardrails than CPU platforms that have been forced to incrementally adjust to a barrage of security attacks. Unfortunately, the peculiarities of the GPU make it difficult to directly port memory safety solutions from the CPU space.

This paper presents cuCatch, a new memory safety error detection tool designed specifically for the CUDA programming model. cuCatch combines optimized compiler instrumentation with driver support to implement a novel algorithm for catching spatial and temporal memory safety errors with low performance overheads. Our experimental results on a wide set of GPU applications show that cuCatch incurs a 19% runtime slowdown on average, which is orders of magnitude faster than state-of-the-art debugging tools on GPUs. Moreover, our quantitative evaluation demonstrates cuCatch's higher error detection coverage compared to prior memory safety tools. The combination of high error detection coverage and low runtime overheads makes cuCatch an ideal candidate for accelerating memory safety debugging for GPU applications.

CCS Concepts: • **Security and privacy → Software security engineering**; • **Computer systems organization → Single instruction, multiple data**; • **Software and its engineering → Runtime environments**.

Additional Key Words and Phrases: memory safety, bug detection, GPU.

ACM Reference Format:
Mohamed Tarek Ibn Ziad, Sana Damani, Aamer Jaleel, Stephen W. Keckler, and Mark Stephenson. 2023. cuCatch: A Debugging Tool for Efficiently Catching Memory Safety Violations in CUDA Applications. *Proc. ACM Program. Lang.* 7, PLDI, Article 111 (June 2023), 24 pages. https://doi.org/10.1145/3591225

1 INTRODUCTION

Programs authored in memory unsafe languages, such as C and C++, are vulnerable to malicious attacks in the worst case, and random crashes in the most benign cases [Song et al. 2019; Szekeres et al. 2013; van der Veen et al. 2012]. Recent studies indicate that common memory related bugs such as buffer overruns and use-after-frees account for roughly two thirds of all exploitable bugs in

Authors' addresses: Mohamed Tarek Ibn Ziad, mtarek@nvidia.com, NVIDIA, Westford, MA, USA; Sana Damani, sdamani@nvidia.com, NVIDIA, Santa Clara, CA, USA; Aamer Jaleel, ajaleel@nvidia.com, NVIDIA, Westford, MA, USA; Stephen W. Keckler, skeckler@nvidia.com, NVIDIA, Austin, TX, USA; Mark Stephenson, mstephenson@nvidia.com, NVIDIA, Austin, TX, USA.

2475-1421/2023/6-ART111
https://doi.org/10.1145/3591225

CPU programs [Google 2019; Miller 2019]. The ramifications of memory safety vulnerabilities in GPU applications are less studied, but recent work suggests that they are also exploitable [Di et al. 2016; Miele 2016; Park et al. 2021].

In response to security vulnerabilities and costly program crashes, the industry has created several memory safety tools and mitigation techniques to identify bugs in CPU programs. Programmers can use software-only debugging tools, such as Google's Address Sanitizer [Serebryany et al. 2012] and Valgrind [Nethercote and Seward 2007], to discover latent bugs during the development process [Song et al. 2019]. The CPU industry is also exploring hardware support to protect memory unsafe languages with low overheads [ARM 2019; Oracle 2015].

While industry and research communities have made significant progress toward improving the safety of C and C++ programs, GPU acceleration platforms have received less attention until recently. NVIDIA's Compute Sanitizer [NVIDIA 2022a] (and cuda-memcheck [NVIDIA 2022c] before it) is the only commercial tool available for memory safety debugging on GPUs. Fortunately, the research community is beginning to take the threat of buggy GPU code seriously. Recent work has investigated both software-only and hardware approaches for hardening GPU programs. The peculiarities of GPU acceleration platforms preclude the trivial adoption of CPU-based solutions, and therefore prior research for GPU memory protection developed tailor-made approaches.

Prior approaches for GPUs are deficient in either their memory safety algorithm or the applied code instrumentation technology. For example, GMOD [Di et al. 2021] and clARMOR [Erb et al. 2017] are software-only approaches that offer low runtime overheads but provide limited error detection coverage. GPUShield proposes new hardware support to limit runtime overheads, but cannot be used on existing GPUs [Lee et al. 2022]. On the other end of the spectrum, NVIDIA's Compute Sanitizer, which relies on dynamic binary instrumentation, requires no hardware modifications and can handle arbitrary GPU binaries, but incurs significant runtime overheads.

In this paper, we present a software-only debugging tool called cuCatch. By combining a novel memory safety algorithm with efficient code instrumentation, we show why cuCatch fills an important gap in the existing tools and research landscape and provides better error detection coverage for GPU compute programs on commodity GPUs. cuCatch provides the following features:

- cuCatch uses a novel algorithm, called *shadow tagged base & bounds* (Shadow TBB), which enables better error detection coverage compared to prior memory safety algorithms, such as canaries, tripwires, and memory tagging.
- cuCatch employs a novel *base pointer analysis* technique to eagerly retrieve the appropriate metadata for performing the memory safety checks.
- cuCatch is implemented in NVIDIA's backend compiler as an instrumentation pass, with the necessary runtime support implemented in the driver. We implement several optimizations to reduce the runtime overheads for detecting memory safety errors in the different GPU memory spaces (i.e., global, local, and shared).
- cuCatch deterministically detects several memory safety violations and probabilistically detects others while incurring low runtime overheads (19% on average).

2 BACKGROUND

2.1 GPU Background

We first describe the high-level characteristics of GPUs and their associated programming models, especially with respect to memory safety. Though the concepts we describe are general to GPU computing platforms, our descriptions use NVIDIA's terminology.

2.1.1 GPU Architecture. GPU programs execute on programmable processing cores called streaming multiprocessors (SMs). A GPU program consists of host-side functions, which run on the

```
1  __global__ void contrived(int *global_arr, int val)
2  {
3      int local_arr[10];  // 24-bit swizzled address.
4      __shared__ shared_arr[10]; // 24-bit address.
5      int *p;  // 64-bit generic address, inferred.
6      switch (val) {
7          case 0: p = global_arr; break;
8          case 1: p = local_arr; break;
9          case 2: p = shared_arr; break;
10     };
11     // p is a 64-bit "generic" pointer.
12     memcpy(p, src, 10*sizeof(int));
13 }
```

(a) Unified addressing example.

Mnemonic	Action
ATOMS	Shared memory atomic
LDS	Shared memory load
STS	Shared memory store
ATOMG	Global memory atomic
LDG	Global memory load
STG	Global memory store
LDL	Local memory load
STL	Local memory store
ATOM	Generic memory atomic
LD	Generic memory load
ST	Generic memory store

(b) Memory operations.

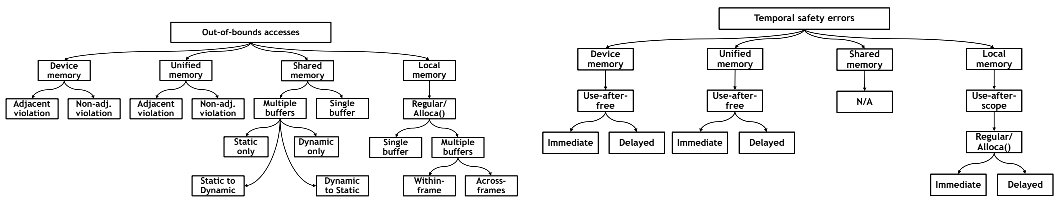
Fig. 1. GPU memory spaces. CUDA allows memory allocation to global, local, and shared memories, and the GPU hardware interprets the addresses of each space differently.

CPU, and device-side functions, which CUDA refers to as kernels. The CUDA programming model allows the creation of thousands of parallel threads, which are grouped into convenient physical entities called warps that span at most 32 threads. The threads in each warp execute in a single instruction, multiple thread (SIMT) fashion to improve efficiency, all fetching from a single Program Counter (PC) in the absence of control flow. In most kernels, many warps map to a single GPU core, or SM. A GPU consists of multiple SMs along with SM-local scratchpad memories and a memory hierarchy consisting of L1 caches, a shared L2 cache, and multiple memory controllers.

2.1.2 GPU Memory Spaces. GPU programming models allow memory allocation within different memory spaces, each of which behaves uniquely. Figure 1a shows a simple kernel that uses *global*, *local*, and *shared* memory. GPU programming models fundamentally handle heap and stack accesses differently. NVIDIA's platforms map heap allocations to the global memory space, which has a 64-bit address space, whereas they map stack allocations to a thread-private 24-bit local address space. While global and local memories are backed by the same memory hierarchy, due to their different access patterns, the underlying hardware operates on global and local memory objects differently. For instance, one thread cannot access another thread's local memory; and to make more efficient use of the memory system in the SIMT programming model, the memory unit swizzles addresses to interleave a warp's local memory accesses so that a (full) warp touches a contiguous 32×4 byte block of data when all threads access the same address on their private stacks.

The *global* memory can be device memory (which can only be accessed from the GPU device) or host-pinned or unified memory (both of which can be accessed from both the GPU and host). Besides *global* and *local* memories, GPU programming models provide a software-managed memory region called *shared* memory, which is shared among threads that belong to the same thread block and execute on the same SM. While *global* memory can be accessed by all threads, it can be orders of magnitude slower than *shared* memory, and as such high-performance programs tend to heavily rely on *shared* memory. As Figure 1b shows, the instruction set architecture (ISA) includes distinct instructions for operating in each space.

To simplify programming in the face of these different memory spaces, CUDA uses a unified address space and *generic* pointers that can point to shared, local, or global memories. Instructions such as ATOM, LD, and ST can act on local, shared, or global memory, depending on where in the unified address space the pointer points. Figure 1a provides an example where the type system cannot infer the space to which p points for the definition at line 12 and therefore allocates a generic pointer for p. The availability of multiple memory spaces on GPUs makes it challenging to enforce



(a) Spatial memory safety vulnerabilities. (b) Temporal memory safety vulnerabilities.

Fig. 2. Different types of memory safety vulnerabilities affecting GPU memory spaces.

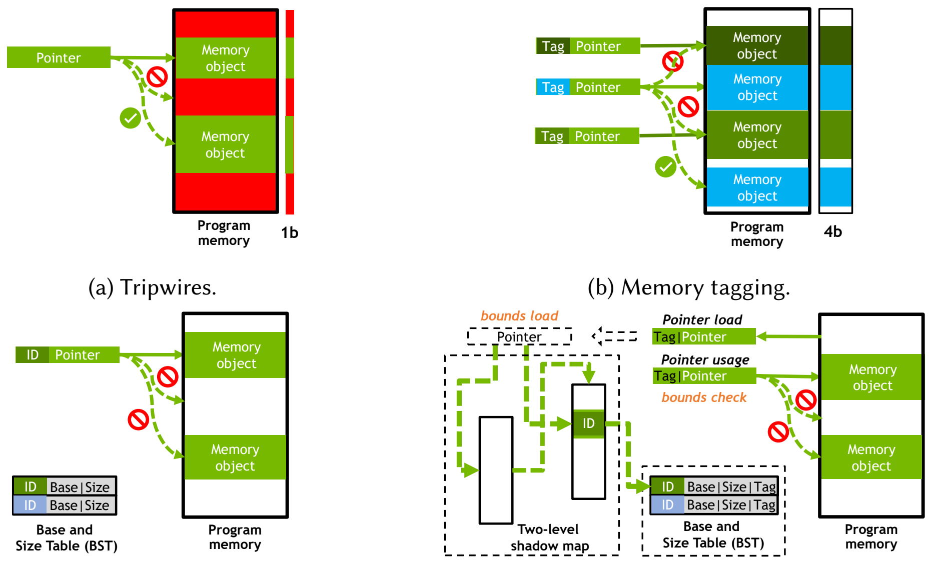
memory safety policies as the maintenance and access of the metadata differ for each memory space. Sections 3 and 4 elaborate on the challenges of protecting heterogeneous memory spaces.

2.2 Memory Safety Background

Memory safety is a major concern for low level programming languages, such as C and C++ [Szekeres et al. 2013; van der Veen et al. 2012], and their extensions, such as CUDA [Di et al. 2016; Miele 2016]. Memory safety is a program property that ensures memory allocations can only be accessed between their intended bounds and during their lifetime [Song et al. 2019]. Violating any of these requirements may result in memory corruption. For example, accessing allocations beyond their intended bounds, such as buffer overruns, violates spatial memory safety. Accessing allocations beyond their lifetimes, such as using an allocation after freeing its memory or reading uninitialized variables, violates temporal memory safety. Given a memory safety vulnerability, attackers can potentially achieve read and write capabilities that can lead to privilege escalation, information leakage, and denial of service on the victim system [Szekeres et al. 2013].

2.2.1 Memory Safety on GPUs. A GPU program can have a spatial and/or temporal memory safety vulnerability. Figure 2 shows different memory safety vulnerabilities affecting GPU memory spaces.
Out-of-bounds access. For device and unified global memory regions, out-of-bounds (OOB) memory safety errors can target nearby allocations (i.e., adjacent) or arbitrary memory locations (i.e., non-adjacent). For shared memory, OOB memory safety errors abuse single and multiple shared buffers, which might be statically allocated as part of the GPU kernel code or dynamically allocated via a kernel launch parameter. Finally, a buffer under-/over-flow can cause local memory OOB accesses within the same stack frame or across different frames. This applies to regular buffers with statically-known sizes and dynamically-allocated local buffers (i.e., using `alloca`).[1]
Temporal safety. We categorize temporal safety vulnerabilities according to the GPU memory space. For example, temporal safety vulnerabilities on device and unified memory are referred to as use-after-free (UAF) whereas temporal safety vulnerabilities on local memory are referred to as use-after-scope (UAS). As the GPU shared memory space cannot be deallocated during kernel execution, it is neither vulnerable to use-after-free nor use-after-scope errors.
Double-free and Invalid-free. Memory safety errors such as double free occur when the same global pointer is freed twice. An invalid free occurs when the memory deallocation API (e.g., `free` or `cudaFree`) is invoked on a pointer that does not point to the starting address of an allocation.
Impact. A memory safety vulnerability in a GPU program may lead to multiple threats. Examples include leaking the secret keys from a GPU-accelerated encryption algorithm or manipulating a self driving vehicle's navigation software. Prior work demonstrated spatial memory safety violations on GPUs by using buffer overflows to overwrite function pointers on the heap and stack memory

[1]`alloca` is a device-side built-in function that can be used to allocate dynamic memory on the stack [Sundaram et al. 2021].



(a) Tripwires.

(b) Memory tagging.

(c) Tagged base & bounds.

(d) Shadow tagged base & bounds.

Fig. 3. Categories of different algorithms used for detecting memory-safety vulnerabilities.

to hijack the program execution flow [Di et al. 2016; Miele 2016]. A recent paper presented an end-to-end GPU attack that degraded the inference stage of a deep learning model [Park et al. 2021]. Starting with a buffer overflow vulnerability in the deep learning code, the authors (1) overwrote a function pointer to control the program execution, (2) injected their own malicious code, and (3) nullified the main decision-making functions in the GPU code pages.

2.2.2 *Memory Safety Algorithms.*

To detect memory safety violations, three main approaches have been previously explored in the literature: tripwires, memory tagging, and base & bounds.

Tripwires (also known as memory blocklisting). As shown in Figure 3a, the key idea is to guard the boundaries of all memory allocations with redzones or tripwires that generate an exception if ever accessed by a load or a store instruction. A popular tripwire-based approach is Google's Address Sanitizer [Serebryany et al. 2012], which maintains a few bits for each byte in memory to decode whether the byte is allocated, freed, or allocated but not initialized. From an error detection standpoint, tripwires are vulnerable to non-adjacent buffer overflows, in which an out-of-bounds violation jumps over the redzones and corrupts data elsewhere without being detected. As per a recent report from Microsoft security research, non-adjacent overflows account for almost one third of the total memory safety vulnerabilities in C and C++ workloads [Bialek et al. 2020].

Memory tagging. Instead of storing the metadata only in memory to mark whether a memory allocation is allocated or not, the metadata is divided between the allocation and the pointer, as shown in Figure 3b. The key idea is to assign a color or tag for each new allocation and to use this color to tag each byte in physical memory that belongs to the allocation. Then, the tag is stored in the upper bits of the pointer that points to the allocation. Every time a memory access occurs, the hardware compares the tag from the pointer to the tag of the accessed memory byte and triggers an exception if they do not match. The rule of thumb here is to assign different colors to adjacent allocations such that buffer overflows and underflows can be easily detected. Examples of memory tagging approaches include SPARC's ADI [Oracle 2015] and ARM's MTE [ARM 2019].

While memory tagging does a great job of catching accidental memory safety violations, its reduced tag size (typically four bits) provides low entropy against non-adjacent OOB accesses as colors will be repeated once the program allocates more than 16 allocations.
Base & bounds (also known as memory permitlisting). To avoid the limitations of memory tagging, base & bounds explicitly stores the base and size information of each memory allocation and looks up this information for every executed load and store to validate the memory access. As the exact base and size are tracked, all overflows including the non-adjacent ones can be detected offering the highest possible error detection coverage across the three approaches. The metadata can be stored in a disjoint table that is indexed either using (1) the pointer location in memory (also known as *shadow base & bounds* [Nagarakatte et al. 2009; Oleksenko et al. 2018]) or (2) an identifier that is stored in the upper pointer bits (also known as *tagged base & bounds* [Lee et al. 2022], which is shown in Figure 3c). Unfortunately, the first approach comes with high metadata maintenance complexity, while the second approach does not scale when the number of live allocations exceeds the size of the disjoint table, which is limited by the number of unused upper pointer bits.

2.3 Goals and Non-goals

2.3.1 Goals. The main goal of this paper is to provide GPU developers with a tool for detecting as many memory safety errors as possible with a low runtime cost. We assume the availability of the parallel thread execution (`ptx` [NVIDIA 2022e]) code, which is an intermediate representation emitted by the front-end compiler after partially compiling the program source code, so that our compiler can analyze it and insert the necessary instructions for accessing the metadata and performing memory safety checks.

2.3.2 Non-goals. While our proposed tool handles programs written in any language that is translated to `ptx` (e.g., `CUDA`, `OpenCL`, or `OpenACC`), we only focus on instrumenting device-side code (thus memory safety errors in the host-side code that runs on the CPU are not reported). Existing CPU tools can be used to handle host-side code [Nethercote and Seward 2007; Serebryany et al. 2012]. Additionally, certain types of memory safety violations, such as uninitialized reads and typecasting between incompatible types, are not covered by our current prototype. Finally, we do not consider synchronization or concurrency bugs, such as race conditions, in this paper as other tools can be used to address them [Kamath and Basu 2021; Peng et al. 2018; Wu et al. 2020].

3 OUR PROPOSED TOOL: CUCATCH

Ensuring memory safety in conventional programs is a two-pronged approach. First, we need a suitable technology to instrument and verify memory addresses accessed by a program. Second, we need a suitable algorithm to detect memory safety violations. Unfortunately, existing memory safety tools on GPUs suffer in their choice of either the memory safety algorithm or the code instrumentation technology. For example, GMOD [Di et al. 2021, 2018] and clARMOR [Erb et al. 2017; Erb and Greathouse 2018] use a low overhead compile-time approach to instrument memory accesses but use a low-coverage memory safety algorithm (i.e., canaries). Similarly, NVIDIA's Compute Sanitizer [NVIDIA 2022a] relies on tripwires, but uses dynamic binary instrumentation to instrument memory accesses, which incurs high performance overheads. Ideally, we desire a code instrumentation technology with low performance overheads and a high-performing scalable memory safety algorithm with high error detection coverage. We propose *cuCatch*, a compile-time memory safety tool that uses a novel memory safety algorithm called *shadow tagged base and bounds* (Shadow TBB) for providing high error detection coverage.

3.1 Overview

Figure 3d provides an overview of our proposed algorithm, Shadow TBB. The key idea is to store the necessary metadata for performing memory safety checks in a disjoint structure called the Base and Size Table (BST). A new BST entry is assigned for each new memory allocation to store the starting address of the allocation, the allocation size, and a random tag. The starting address and size entries in the BST are both self-explanatory, while the *tag* is a random value assigned to each new allocation to detect temporal memory safety violations. Given any arbitrary pointer, we need to identify the BST entry that belongs to the allocation. If the BST-entry index can fit in the upper pointer bits, it will be directly embedded in the pointer. Otherwise, we establish the connection between pointers and their corresponding BST entry by using a *shadow map*. The shadow map is a page-table like data structure that stores an N-bit BST-entry index for each M-byte virtual memory region. To retrieve the BST-entry index of a given pointer, we use the pointer value to index into a two-level shadow map (see Figure 3d).

3.2 Allocation Life-cycle

3.2.1 Allocation Creation. When a new allocation is created (e.g., by calling a memory management API such as `cudaMalloc`), cuCatch stores the allocation base address and size in a new BST entry and assigns a random non-zero 4-bit value for the tag. Depending on the number of unused upper pointer bits, x, we subdivide the x-bit pointer tag space as follows:[2] $\{0\}$ is reserved for host-side allocations, $[1:15]$ is used for traversing the shadow map, and $[16:(2^x-1)]$ is used to directly index the BST. As a result, for the first (2^x-16) allocations of the program, the BST-entry index is directly stored in the upper pointer bits, allowing for fast retrieval of the allocation metadata. When the number of allocations exceeds (2^x-16), cuCatch first stores the 4-bit random tag (i.e., $[1:15]$) into the upper pointer bits and then stores the full 32-bit BST-entry index inside all the shadow map entries that belong to this allocation. For example, assuming a shadow map that maintains 32-bit integers per each 32-byte virtual memory region, a 512-byte allocation will use $\frac{512}{32}=16$ shadow map entries where all entries hold the same 32-bit BST-entry index.

3.2.2 Pointer Load. During program execution, pointers can be loaded from memory and subsequently used as addresses for memory instructions. Upon loading a pointer from memory, cuCatch retrieves the metadata associated with the pointer by either (1) reading the upper bits of the pointer to determine the BST-entry index (i.e., when the upper pointer bits hold a value in the $[16:(2^x-1)]$ range) or (2) using the pointer value to index into the shadow map to retrieve the BST-entry index (i.e., when the upper pointer bits hold a value in the $[1:15]$ range). The BST-entry index is then used to consult the BST to retrieve the base, size, and tag information associated with the pointer. This metadata is then stored in registers and propagated to all memory instructions that use this pointer as an operand. In doing so, cuCatch effectively constructs a fat pointer that holds all the memory safety information without changing the memory layout [Watson et al. 2015] or application binary interface [Nagarakatte et al. 2009].

3.2.3 Pointer Usage. When a pointer is used as the address operand for a memory instruction, cuCatch inserts a memory safety check before the actual memory access. The memory safety check uses the resulting pointer value (and tag) of the memory instruction and the metadata information (which was previously fetched from the BST) to determine whether the memory access is legal or not. cuCatch flags an exception if (1) the memory access is not within the legitimate allocation bounds or (2) there is a temporal tag mismatch.

[2]The unused upper pointer bits depend on the host CPU architecture and page size used. They typically range from 7 to 16 bits. In our current prototype, we assume $x = 8$ bits.

3.2.4 Allocation Deletion. When an allocation is deleted by invoking a memory management API call such as `cudaFree`, cuCatch sets the corresponding tag in the BST entry to zero such that subsequent accesses to the memory allocation via a dangling pointer will be detected. In addition, cuCatch zeros the allocation's shadow map entries so that they point to the illegal, zeroth BST entry. New allocation requests always receive a fresh BST entry (and a random, non-zero 4-bit tag if they are accessed through the shadow map). Unlike prior work [Lee et al. 2022], our BST size is not constrained by the number of currently unused upper pointer bits.

3.3 Benefits of the Shadow TBB Approach over Prior Memory Safety Algorithms

Shadow TBB is a memory safety algorithm that tracks the precise bounds of each allocation. As such, it offers higher error detection coverage than canaries (which is used by GMOD and clARMOR) and tripwires (which is used by NVIDIA's Compute Sanitizer). Shadow TBB draws inspiration from prior *base & bounds* approaches (namely Softbound [Nagarakatte et al. 2009] and GPUShield [Lee et al. 2022]) and from memory tagging (namely ARM's MTE [ARM 2019]). SoftBound is a compiler-based memory safety tool for CPUs that stores the base and size information in a shadow memory table that is indexed using the pointer's location in memory (i.e., pointer's address). On the other hand, GPUShield is a hardware-assisted memory safety mechanism for GPUs. GPUShield uses a *tagged base & bounds* (TBB) algorithm, which stores the base and size information in a disjoint table that is indexed using a tag that is embedded in the upper bits of a pointer. Shadow TBB takes the best features of SoftBound, GPUShield, and ARM's MTE while addressing their limitations.

GPUShield limits the scalability of the memory safety algorithm due to the limited number of unused bits in a pointer [Lee et al. 2022]. If a GPU program uses more than 2^x memory allocations, a TBB-only approach does not scale. Shadow TBB addresses this issue by using a traditional TBB approach for the first $(2^x - 16)$ device-side allocations then falls back to a shadow map approach for covering the remaining allocations, including managed (i.e., unified) memory.

SoftBound's shadow metadata table, which is indexed using the pointer's address, complicates the metadata maintenance [Nagarakatte et al. 2009]. The compiler must copy the base and bounds information in memory when a pointer is copied from one place to another (e.g., copying an array of pointers using `memcpy`). If the compiler does not intercept a single pointer-copying site, SoftBound's shadow metadata table will become incomplete (i.e., the copied pointer will have no associated metadata when it is used later to access memory). Incomplete shadow metadata tables lead to false alarms, which negatively affect the usability of a debugging tool. This scenario is relevant to GPU applications because host-side code and opaque third-party libraries such as cuFFT and cuDNN, can copy around memory chunks, including pointers. Shadow TBB avoids these challenges by indexing the shadow map using the pointer value. As a result, no special treatment is needed for pointer copying as pointer location is irrelevant to the metadata indexing. Further, this design choice better serves the debugging goals of cuCatch, in which there are no active attackers who try to force out-of-bounds pointers to escape to memory before loading them in a different context.

4 IMPLEMENTATION

cuCatch is implemented in a recent fork of a production CUDA toolkit. This section describes our changes to the CUDA driver and the production GPU-compute backend compiler. As Figure 4 illustrates, compilation begins with a high-level input specification of a CUDA program. A *front-end* compiler lowers the program's specification to an intermediate representation called PTX (parallel thread execution). We modified the backend *ptxas* compiler to generate an instrumented application, which relies on a modified CUDA runtime. We first describe our compiler-based analyses, instrumentation, and optimizations. We then discuss the role of the runtime, which is responsible for maintaining the metadata structures that the instrumented code consults.

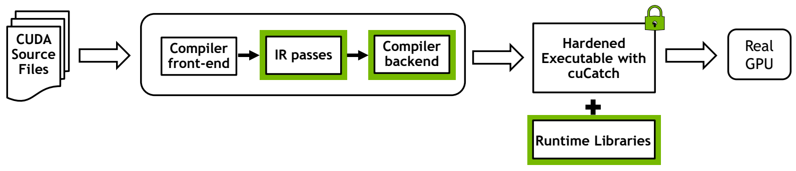


Fig. 4. The cuCatch infrastructure overview.

4.1 Compiler Instrumentation

Compiler instrumentation is a two-step process. In the analysis step, which we call *base pointer analysis*, we slice backwards from memory instructions through pointer arithmetic to try to identify the minimal set of pointers from which all other pointers are derived. In the transformation step, we insert code that fetches the metadata for each identified root pointer. For global memory allocations (also referred to as buffers), this step involves accessing the base, bounds, and tag information stored in the BST populated by the driver. For shared and local memory buffers, the compiler fetches array bounds information from the symbol table populated by the front-end compiler. Finally, we insert memory safety checks that use this metadata information to ensure spatial and temporal safety before each memory access in the program. We describe these steps below.

4.1.1 Base Pointer Analysis. To ensure that we fetch metadata for a buffer only once, and we do not use an out-of-bounds address to fetch metadata, we perform *base pointer analysis*, a dataflow analysis inspired by the well-known reaching definitions analysis [Aho et al. 2007] that computes the reaching base pointers for each memory access.

Definition: A base pointer definition "S: p(64b) = ..." at a program point u reaches a program point v if there is some path from u to v along which there is no *destructive redefinition* of p.

We define a destructive redefinition as a definition of p such that p now potentially points to a new object in memory. Examples of destructive redefinitions include loads from memory, function calls, and arithmetic instructions with multiple reaching base pointers. As with reaching definitions, base pointer analysis is an iterative, forward dataflow analysis that builds *baseptrdef-use* chains. We show the dataflow equations for our base pointer analysis below.

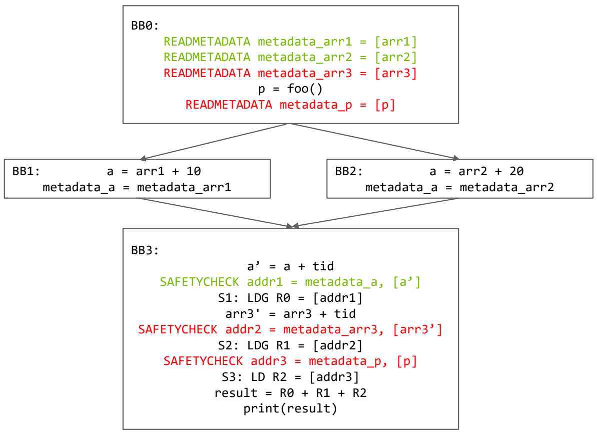
$$\begin{aligned}
\text{GEN(BB)} &= \text{Set of downward-exposed destructive pointer definitions in BB}\\
\text{KILL(BB)} &= \text{Destructive redefinitions of pointers in IN(BB)}\\
\text{IN(BB)} &= \cup_{P \in predecessors(BB)} \text{OUT(P)}\\
\text{OUT(BB)} &= (\text{IN(BB)} - \text{KILL(BB)}) \cup \text{GEN(BB)}
\end{aligned}$$

For the example in Figure 5, our base pointer analysis correctly identifies `arr1` and `arr2` as the reaching base pointers for memory access `S1` in `BB3`. On the other hand, it identifies `obj3+offset` as the base pointer that reaches `S2` as the device-side compiler cannot analyze host-side code that calls the kernel. Furthermore, the compiler cannot analyze function `foo()` and identifies its return value `p` as the reaching base pointer for `S3`, which may not be true if `p` was the result of pointer arithmetic. While identification of the true base pointer is useful for efficiency and coverage, failure to identify the base pointer does not always preclude identification of memory safety errors, as we discuss in Section 7. After our analysis has identified the base pointers, cuCatch's compiler pass performs code instrumentation.

4.1.2 Instrumentation. Table 1 shows our additions to the IR to support memory safety instrumentation. Since these instructions are not natively supported by the GPU hardware, our compiler

```
Kernel(int8 *arr1, int8 *arr2, int8 *arr3)
{
    p = foo();
    if (condition) {
        a = arr1 + 10;
    } else {
        a = arr2 + 20;
    }
    print a[tid] + arr3[tid] + *p;
}

Main()
{
    Kernel<<<...>>>(obj1, obj2, obj3+offset);
}
```



```
BB0:
    READMETADATA metadata_arr1 = [arr1]
    READMETADATA metadata_arr2 = [arr2]
    READMETADATA metadata_arr3 = [arr3]
    p = foo()
    READMETADATA metadata_p = [p]

BB1:
    a = arr1 + 10
    metadata_a = metadata_arr1

BB2:
    a = arr2 + 20
    metadata_a = metadata_arr2

BB3:
    a' = a + tid
    SAFETYCHECK addr1 = metadata_a, [a']
    S1: LDG R0 = [addr1]
    arr3' = arr3 + tid
    SAFETYCHECK addr2 = metadata_arr3, [arr3']
    S2: LDG R1 = [addr2]
    SAFETYCHECK addr3 = metadata_p, [p]
    S3: LD R2 = [addr3]
    result = R0 + R1 + R2
    print(result)
```

Fig. 5. Example showing the additional intermediate representation (IR) instrumentation added by cuCatch.

Table 1. The new IR instructions used by cuCatch.

Instruction	**Description**
READMETADATA metadata = [base]	Returns base and bounds.
OOBCHECK base, bounds, [addr]	Checks if address lies between base and base+bounds.
TAGCHECK newAddr = base, metadata, [addr]	Performs tag read & check for temporal safety & returns untagged address.
SAFETYCHECK newAddr = base, metadata, [addr]	Performs the operations of both OOBCHECK and TAGCHECK.
MASK newAddr = [addr]	Returns an untagged address.

emulates the new instructions. The code that our compiler emits to emulate these high-level instructions contains conditional control flow instructions, which can potentially degrade performance, as will be studied in Section 6.2.4.

The goal of the instrumentation is to convert base pointers into fat pointers which we propagate through the kernel. For each identified base pointer, the instrumentation pass inserts READMETADATA instructions that return pointer metadata from the BST. Further, the pass inserts SAFETYCHECK instructions that use the metadata to flag memory safety violations. Figure 5 shows that the pass inserts a safety check instruction before each memory access to check whether the supplied address lies within the bounds of the corresponding array (spatial safety), and also ensures that the tag matches (temporal safety) before masking off the tag bits and returning the original address. If the safety check instruction identifies an out-of-bounds access or a use-after-free, it generates an error and program execution ends.

```
p = …
while(condition)
{
    LDG[p]
    p = p + k
}
```

(a) Original program.

```
p = …
while(condition)
{
    READMETADATA metadata = [p]
    SAFETYCHECK addr = metadata, [p]
    LDG[addr]
    p = p + k
}
```

(b) Naive instrumentation.

```
p = …
READMETADATA metadata = [p]
while(condition)
{
    SAFETYCHECK addr = metadata, [p]
    LDG[addr]
    p = p + k
}
```

(c) cuCatch instrumentation.

Fig. 6. cuCatch's base pointer analysis removes redundant metadata fetch instructions.

Figure 6a shows a simple while loop that reads elements from an array. Figure 6b shows a naively instrumented program with metadata reads and safety check instructions inserted before each memory access. This version of the instrumented program suffers from efficiency and correctness issues. First, the metadata read within the loop is redundant because the metadata information for a given buffer remains constant for the entire duration of the program. Second, if pointer `p` goes out of bounds of the current buffer and instead points to a different, valid buffer, this version will fail to identify the overflow between buffers. Figure 6c shows that as a consequence of our base pointer analysis, cuCatch only inserts metadata fetch instructions once for the base pointer and reuses the metadata whenever an instruction accesses the same buffer. This approach leads to both error detection and performance improvements over the naive solution.

4.2 Shared Memory Protection

Our prototype uses neither a BST nor shadow memory for shared memory addresses but instead relies heavily on static analysis. In most kernels the compiler can effectively infer the space of shared memory operations, and furthermore, it can infer shared memory base pointers. CUDA supports two types of shared memory allocations: static and dynamic [Harris 2013]. Static arrays are declared in kernel code, and the front-end compiler passes the size information to the backend compiler. However, dynamic arrays are declared in host code and are coalesced into a single buffer allocated after static arrays. We discuss the error detection consequences of this behavior in Section 5.2. As shared memory cannot be deallocated during kernel execution, we do not use tags to catch temporal safety violations. Hence, we use a bounds check instruction (`OOBCHECK`) that only checks if the address lies within the allocation's base and bounds without checking the tag or masking off any bits from the address. We still use the base pointer analysis described in Section 4.1.1 to identify reaching base pointers. Furthermore, because there is no BST and the metadata (base and bounds) is statically known, we do not insert a metadata read instruction for shared memory accesses. This makes shared memory error checking much cheaper than global memory error checking, both in terms of register usage and memory accesses.

4.3 Local Memory Protection

Like cuCatch's global memory support, we record stack allocations in a BST and tag pointers (i.e., store the BST-entry index in the pointer's upper bits) to local memory. However, the thread-private nature of local memory (e.g., two threads that load from the address `0xffffa0` touch different regions in the generic address space) along with its limited address space warrants a customized BST. Instead of storing a monolithic BST to record all local memory allocations, cuCatch maintains a per-thread 31-entry BST in local memory. CUDA's current specification allows for a per-thread 16 MB stack, which means an address in local memory can be at most 24-bits, leaving 8 bits of tagging space that we partition into a 5-bit local index that serves as a reference into the per-thread BST, and a 3-bit local "tag" that we use to identify temporal safety issues on the stack.

In the header of each kernel or function with a stack frame, cuCatch emits code that initializes a BST entry for the frame. The per-thread BST itself is implemented as a stack. When a function with a frame executes, our emitted code pushes details of the stack frame (i.e., its base, bounds, and a random 3-bit tag) onto the BST, and when the frame unwinds, our emitted code pops the BST entry. We pack the upper eight bits of the frame's stack pointer with the BST reference and the random tag. Our support for `alloca` is similar, with the exception that unwinding the frame can pop multiple BST entries as multiple `alloca` buffers can coexist in a single stack frame.

Other than the differences specified above, the general approach between our local memory and global memory support is similar: memory safety checking unpacks the BST reference and the tag stored in the pointer and then references the BST entry to compare the tags and ensure that the reference is in bounds. A 3-bit temporal safety tag allows us to catch temporal safety issues, on a per-thread basis, with a probability of $1 - \left(\frac{1}{2^3}\right) = \frac{7}{8}$. However, if we consider the SIMT nature of most CUDA programs, the probability that we identify temporal issues can be much higher. With a per-thread random tag we can model the probability of catching a temporal safety violation on the stack as $p = 1 - \left(\frac{1}{2^3}\right)^n$, where n is the number of active threads in a warp and is often 32.

If pushing a frame's BST entry would exceed the space the driver reserved for storing the thread-local BST, cuCatch will not tag the stack pointer. Our prototype treats untagged local memory pointers specially and reverts to the CUDA compiler's current `-sp-bounds-check` functionality, which simply ensures that stack pointers do not overrun their allocated space. These protections, combined with the fact that deep call graphs are extremely uncommon in CUDA applications, limit the risk of stack overflows. In addition to a limited BST size, the backend compiler in which cuCatch operates does not have visibility into the contents of a frame, so our prototype cannot detect intra-frame overruns; but this is a technicality of our prototype and not a flaw in the algorithm.

4.4 Handling Generic Pointers

When the compiler cannot infer the space to which an address points, it uses *generic* memory operations, as explained in Section 2.1.2. We protect generic memory instructions by emitting code that first disambiguates the memory space (using the CUDA functions `__isShared()`, `__isLocal()`, and `__isGlobal()`) before branching to code that appropriately unpacks the generic address's tag and implements the protection mechanism for the associated memory space.

Generic addresses also require cuCatch to instrument conversions between memory spaces. When a local address is converted to a generic address, via `__cvta_local_to_generic()`, cuCatch emits code that transfers the local address's 8-bit tag to the resultant generic address's tag space. Conversions from a global address to a generic address require no special treatment.

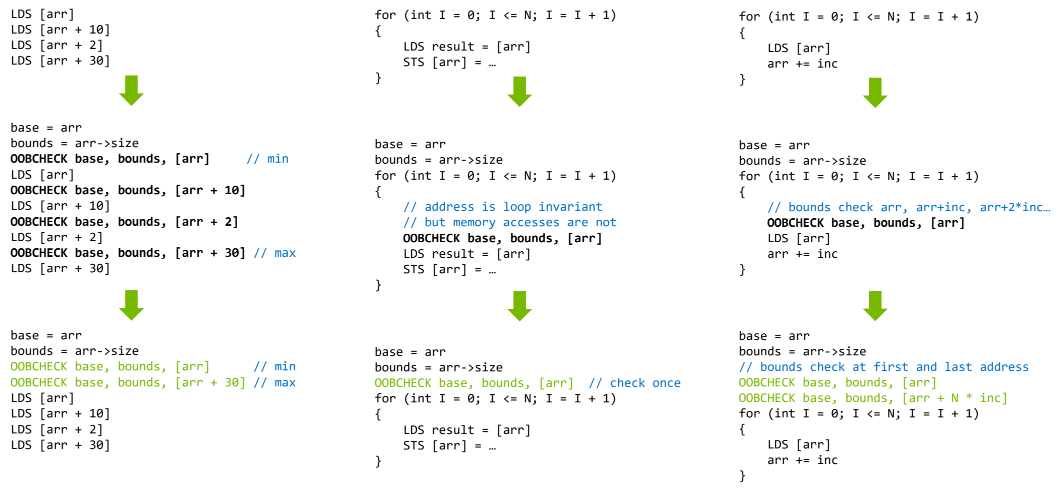
4.5 Error Attribution

cuCatch provides a compile-time knob for controlling the behavior of memory safety exceptions. In one mode, cuCatch simply executes a `trap` instruction that terminates the program. This mode is useful for allowing cuCatch to run in the context of a debugger like `cuda-gdb`. Another mode provides standalone error reporting: Before forcing application shutdown, cuCatch displays the failure reason, including the offending memory address, the allowable boundaries of the allocation, and PTX line information. The latter mode incurs additional overheads due to increased register usage which can reduce a kernel's effective parallelism.

4.6 Optimizations

In Section 4.1, we described the problem of redundant metadata reads. We now detail cases where the bounds check may also be redundant.

4.6.1 Coalesce Bounds Checks to the Same Memory Allocation within Straight-line Code. Figure 7a shows straight-line code with multiple accesses to the same memory allocation (in this case, a shared memory buffer, `arr`). The naive instrumentation inserts a bounds check instruction before every such memory access to `arr`. We propose an optimization that walks over all accesses to the same buffer and identifies the minimum and maximum address accessed. Next, we check if the minimum and maximum addresses both lie within the bounds of `arr` before executing any of the memory reads. This optimization has two benefits. First, it reduces the number of bounds checks thereby reducing the total computations for shared memory accesses and the total number of memory accesses in the case of global memory accesses. Second, because we no longer insert



```
LDS [arr]
LDS [arr + 10]
LDS [arr + 2]
LDS [arr + 30]

base = arr
bounds = arr->size
OOBCHECK base, bounds, [arr]      // min
LDS [arr]
OOBCHECK base, bounds, [arr + 10]
LDS [arr + 10]
OOBCHECK base, bounds, [arr + 2]
LDS [arr + 2]
OOBCHECK base, bounds, [arr + 30] // max
LDS [arr + 30]

base = arr
bounds = arr->size
OOBCHECK base, bounds, [arr]      // min
OOBCHECK base, bounds, [arr + 30] // max
LDS [arr]
LDS [arr + 10]
LDS [arr + 2]
LDS [arr + 30]
```

(a) Bounds check for min and max access to array.

```
for (int I = 0; I <= N; I = I + 1)
{
    LDS result = [arr]
    STS [arr] = …
}

base = arr
bounds = arr->size
for (int I = 0; I <= N; I = I + 1)
{
    // address is loop invariant
    // but memory accesses are not
    OOBCHECK base, bounds, [arr]
    LDS result = [arr]
    STS [arr] = …
}

base = arr
bounds = arr->size
OOBCHECK base, bounds, [arr]  // check once
for (int I = 0; I <= N; I = I + 1)
{
    LDS result = [arr]
    STS [arr] = …
}
```

(b) Hoist loop invariant bounds checks.

```
for (int I = 0; I <= N; I = I + 1)
{
    LDS [arr]
    arr += inc
}

base = arr
bounds = arr->size
for (int I = 0; I <= N; I = I + 1)
{
    // bounds check arr, arr+inc, arr+2*inc…
    OOBCHECK base, bounds, [arr]
    LDS [arr]
    arr += inc
}

base = arr
bounds = arr->size
// bounds check at first and last address
OOBCHECK base, bounds, [arr]
OOBCHECK base, bounds, [arr + N * inc]
for (int I = 0; I <= N; I = I + 1)
{
    LDS [arr]
    arr += inc
}
```

(c) Coalesce bounds checks to same array within a loop.

Fig. 7. cuCatch's optimizations for removing redundant bounds checks. The top row shows the original code followed by the naive and optimized cuCatch instrumentation in the middle and bottom rows, respectively.

conditional branches before each LDS instruction, the instruction scheduler is better able to reorder and group memory access instructions for higher instruction level parallelism.

4.6.2 Hoist Loop Invariant Bounds Checks. Figure 7b shows a `for` loop which accesses the same element of buffer `arr` in every iteration. However, since there is also a write to the same element, the memory accesses are not loop invariant and cannot be hoisted out of the loop. The bounds check instruction on the other hand is loop invariant. Once we know that the address is within bounds, it will remain within bounds for every subsequent iteration. Hence, it is legal to hoist the bounds check instruction out of the loop using loop invariant code motion [Aho et al. 2007]. Once again, this approach reduces the number of computations, memory accesses and improves instruction scheduling, particularly if the loop is unrolled.

4.6.3 Coalesce Bounds Checks to the Same Array across Loop Iterations. Figure 7c shows a similar `for` loop, but this time the LDS strides over multiple elements of the array. Hence, the bounds check is no longer loop invariant because while `arr+inc` may be within bounds, `arr+2*inc` may be out of bounds. Instead, we identify the minimum and maximum addresses accessed across all iterations of the loop and hoist them out of the loop body to ensure all accessed elements lie within bounds before executing the loop. Note that we lose debugging precision when a coalesced bounds check instruction fails because it might refer to any one of multiple memory accesses.

4.6.4 Temporal Safety and Optimizations. Figure 7 summarizes how we reduce redundant bounds checks for shared memory accesses. These optimizations also apply to global memory accesses, with a few modifications. Since global memory buffers may be freed at any time during kernel execution, eagerly performing coalesced temporal safety checks may cause us to miss use-after-free errors later in the program. To ensure cuCatch still detects temporal safety errors, we only hoist or eliminate redundant bounds check instructions and replace deleted OOBCHECKs with TAGCHECKs. We further optimize tag checks by only performing a tag check before the final access to a global object to assert that all previous accesses were safe, which can eventually catch use-after-free errors.

4.7 Runtime Support

The CUDA driver provides support for managing the CUDA runtime APIs, which include all the functions for memory management in CUDA except for thread-level stack management, which the compiler orchestrates. Before a kernel launches, the driver acquires the device resources necessary for the kernel's entire lifetime, including registers, requested stack space, and shared memory.

For this work we modified the CUDA driver by interposing the memory management routines with memory safety functionality. Broadly speaking, we classify memory management routines into two main categories: device memory routines (e.g., `cudaMalloc` and `cudaFree`), which manage memory that is only accessible on the GPU,[3] and unified memory routines (e.g., `cudaMallocManaged` and `cudaHostAlloc`), which manage memory that is accessible from both the GPU and the CPU. We make this distinction because cuCatch cannot tag pointers that reference unified memory, otherwise the pointers would be invalid on architectures that do not support "Top Byte Ignore" (TBI)[4] including the current generation of Intel CPUs.

4.7.1 Device Memory. The cuCatch driver intercepts device memory routines, and after allocating the memory in device RAM, tags the resultant pointer and modifies the BST according to the algorithm presented in Section 3. The BST is allocated per CUDA context, and by default can accommodate 2^{20} live allocations. Whenever the cuCatch driver modifies the BST, it instructs the GPU to flush all caches before returning control to the user's code. When the number of allocations exceeds $(2^x - 16)$, cuCatch uses a random 4-bit value to decorate the upper pointer bits and stores the allocation's BST-entry index in the shadow map. The shadow map is currently implemented as a two-level lookup map, inspired by Valgrind [Nethercote and Seward 2007]. Every time device memory is allocated or freed, cuCatch must modify the BST, which incurs a fixed one-time cost.

4.7.2 Unified Memory. For unified memory allocations, cuCatch reserves a BST entry and populates it with the allocation's details. However, since cuCatch cannot tag the pointer, it uses the shadow map approach described in Section 3.2.1 to store the allocation's BST-entry index without the random 4-bit temporal safety tag. Frees to unified memory allocations modify the BST and the shadow map appropriately. The cuCatch driver also interposes the implicit allocations for global variables, which can map to either device or unified memory, depending on programmer annotations. In our current implementation, every time unified memory objects are allocated or freed, cuCatch incurs an overhead proportional to the size of the allocation to modify the shadow map.

4.7.3 Shared and Local Memory. Before the kernel launches, the cuCatch driver populates the extent of shared memory in a constant bank that our compiler-generated instrumentation code consults. We also reserve thread-private local memory that cuCatch uses to manage the local-memory BST.

5 ERROR DETECTION ANALYSIS

We now analyze cuCatch error detection coverage and compare it to other GPU tools.

5.1 Memory Safety Benchmarks

A GPU program under-test could have one or more memory safety errors. To quantify the error detection coverage of cuCatch, we created a CUDA-based benchmark suite that covers different types of memory safety vulnerabilities (Figure 2). Specifically, our suite includes 56 tests covering fine- and coarse-grained spatial and temporal memory safety violations that can occur in the global, local, and shared address spaces of a GPU.

[3]Our prototype does not yet support the in-kernel memory management APIs (e.g., `malloc` and `free`) [NVIDIA 2023a] because they are rarely used in practice.

[4]TBI is a hardware feature introduced with ARMv8 that ignores the top byte of a pointer when accessing memory.

Table 2. Error detection benchmarks summary.

Benchmark Type	Total Tests	Number of detected tests per tool				
		Baseline	Compute Sanitizer [NVIDIA 2022a]	GMOD* [Di et al. 2018]	GPUShield* [Lee et al. 2022]	cuCatch
Global memory OOB	8	0	4	4	8	8
Local memory OOB	16	0	4	0	12	12
Shared memory OOB	12	0	4	0	0	10
Intra-allocation OOB	8	0	0	0	0	0
Use-after-free	4	0	2	0	0	2
Use-after-scope	4	0	2	0	0	4
Invalid free	2	2	2	2	2	2
Double free	2	2	2	2	2	2
Detection Rate	56	7.1%	35.7%	14.2%	42.8%	71.4%

* Results for GMOD and GPUShield are estimated based on each paper's description.

Table 2 summarizes the results of running our vulnerability suite on different GPU-based memory safety tools using the experimental setup from Section 6.1. The "Baseline" column corresponds to running the vanilla tests on a commodity GPU whereas the "Compute Sanitizer" column corresponds to running the tests with NVIDIA's Compute Sanitizer memcheck tool. As we could not run the tests on an actual system with the necessary modifications for GMOD [Di et al. 2021, 2018] and GPUShield [Lee et al. 2022], we estimated their error detection results as per each paper's description.[5] The final column shows the results of compiling and running the tests with our cuCatch toolchain. Our quantitative results show that cuCatch offers the highest error detection coverage.

5.2 Out-of-bounds Accesses Analysis

5.2.1 Global Memory Coverage. To analyze out-of-bounds (OOB) accesses on global memory, we use a total of eight tests. Compute Sanitizer and GMOD failed to detect four tests that correspond to non-adjacent buffer over-read/write because both tools implement the tripwires algorithm, which can only catch linear overruns. On the other hand, all tests were correctly reported by GPUShield and cuCatch as both techniques track the exact bounds of each allocation. Thus illegal accesses from one valid allocation to another non-adjacent allocation are detected.

5.2.2 Local and Shared Memory Coverage. To analyze OOB accesses on local and shared memory, we run 16 and 12 tests, respectively, as both memory spaces have multiple subcategories as shown in Figure 2a. Compute Sanitizer only captures errors that go beyond the stack (or shared) memory bounds. It misses the tests where overruns occur within the same stack frame, across different frames, and between multiple (statically- or dynamically-allocated) shared buffers. As GMOD only considers global memory, it misses all shared and local memory tests. Further, the GPUShield paper only describes techniques for guarding global and local memory without explaining how shared memory buffers are treated. On the other hand, cuCatch provides significantly higher memory safety coverage by successfully catching 12 and 10 OOB errors on local and shared memory, respectively. cuCatch failed to catch six tests which we describe next.

5.2.3 OOB Errors that cuCatch Misses. cuCatch failed to detect (1) four local memory tests that correspond to adjacent and non-adjacent OOB read/write errors within the same frame, and (2) two shared memory tests that allocate multiple shared buffers from the dynamically-allocated pool. In the above cases, the compiler backend cannot infer the exact bounds of each individual buffer as (1) the precise bounds information of statically-allocated local buffers is unavailable in the

[5] GPUShield requires hardware changes unavailable on today's GPUs and GMOD source code fails to compile with our tests.

compiler backend, and (2) dynamically-allocated shared buffers are reserved as one region at kernel launch time. cuCatch opts to provide coarse-grained memory safety checks to ensure that the OOB accesses do not affect buffers in other memory spaces. Supporting fine-grained bounds checking for dynamically-allocated shared buffers can be achieved with user-annotation. On the other hand, fine-grained bounds checking for statically-allocated local buffers can be achieved by passing more metadata from the compiler front-end to the backend. We leave these extensions to future work.

Finally, all tools failed to detect intra-allocation OOB errors that occur between two fields within the same structure. Those tests go undetected as current tools only track the bounds of the entire allocation (e.g., struct) and lack visibility into the allocation's underlying structure.

5.3 Temporal Safety Analysis

5.3.1 Global Memory Coverage. We test immediate and delayed use-after-free (UAF) errors affecting both device and unified memory. When a memory region is immediately accessed with a dangling pointer after being freed (and before being assigned to a new allocation), cuCatch successfully reports a temporal safety error as the BST entry of the freed region is set to a unique value.

On the other hand, the memory allocator can reuse the recently freed memory region to serve a new allocation request (and before the dangling pointer is used to access the region); we refer to this case as a delayed UAF,[6] which results in two different scenarios. First, if the freed region is assigned a tag in the $[16 : (2^x - 1)]$ range, the violation is deterministically detected by cuCatch due to a mismatch between the BST-entry index of the dangling pointer, which is stored in its upper bits, and the BST-entry index of the new allocation.[7] Second, if the freed region is assigned a tag in the $[1 : 15]$ range (with a shadow map entry), the violation is probabilistically detected because while both the dangling pointer and the new allocation retrieve the same value from the shadow map, they might have been assigned different random 4-bit tags. Compute Sanitizer only detects the immediate UAF errors. We exclude GMOD and GPUShield from our temporal safety analysis as they only focus on spatial memory safety.

5.3.2 Local Memory Coverage. Similar to UAF errors on device and unified memory, temporal safety violations might occur on the stack. We refer to this case as use-after-scope (UAS), which occurs due to accessing a local buffer after its stack frame is destroyed (i.e., after the corresponding function returns). As explained in Section 4.3, cuCatch's per-thread BSTs allow us to deterministically detect immediate UAS errors and probabilistically capture delayed UAS errors. Compute Sanitizer only detects the immediate UAS violations.

5.3.3 Temporal Safety Errors that cuCatch Misses. To summarize, cuCatch deterministically detects all immediate temporal safety errors. On the other hand, cuCatch's ability to detect delayed temporal safety errors comes from the upper pointer bits. As the upper pointer bits store the non-zero 4-bit random tag (i.e., shadow map allocations), the error detection is probabilistic. If the upper pointer bits are not tagged (i.e., unified memory without TBI), the delayed temporal error is not detected.

5.4 Double- and Invalid-free Analysis

Our prototype does not explicitly track allocated and deleted pointers. We rely on the underlying memory allocator (e.g., the GPU allocator that implements `cudaMalloc`) for doing so. Upon passing an invalid (i.e., non-base) pointer to the deallocation function or freeing the same allocation twice, the memory allocator returns a runtime exception, which our tool forwards to the user. The same errors are observed while experimenting with the baseline and NVIDIA's Compute Sanitizer.

[6]Delayed UAF is also known as use-after-realloc (UAR).

[7]If the new allocation is assigned the same BST-entry index as the freed object, the error detection becomes probabilistic.

6 PERFORMANCE EVALUATION

This section analyzes the performance and memory overheads of cuCatch on GPU applications. We first describe our evaluation methodology and workloads and then discuss the runtime results.

6.1 Evaluation Methodology

We evaluate cuCatch using a recent build of the CUDA toolkit, where we replace both the backend `ptxas` compiler with our cuCatch-enabled version, and the CUDA runtime library with a specialized version that provides the runtime support discussed in Section 4.7.

We use 87 standalone CUDA kernels from various workload segments such as scientific computing (e.g., `namd`, `amber18`, `AMG`, `FUN3D`, `Laghos`, `lammps`, `Relion`), commercial (e.g., 5G decoding), and visualization (e.g., Optix [Parker et al. 2010]). We also evaluate PolyBench-ACC [Grauer-Gray et al. 2012] and most of the CUDA Samples [NVIDIA 2022d].[8] Of the total 176 workloads evaluated, only five have 240 (which is $2^x - 16$ for $x = 8$) or more live memory allocations.

We use an NVIDIA GeForce RTX 2080Ti GPU, locked to base clock settings of 1710 MHz for the GPU core and production DRAM frequency settings. The host system is a 12-core Intel Core i7 – 5930K CPU. For the CUDA Samples and PolyBench-ACC, we compare wall clock time from the built-in performance reporting of each application. For the standalone kernels, we use nsight-compute [NVIDIA 2023b] to capture wall clock time for the GPU-only portion of the CUDA kernels, not including kernel launch overheads. As such, these are worst-case results because we ignore execution that can dilute the overheads in short-running kernels, such as initializing constant banks and launching kernels. We do not include the performance overhead of intercepting memory management routines and updating cuCatch's data structures (e.g., BST or shadow map) as these overheads are typically negligible compared to overall application execution time.

6.2 Evaluation Results

6.2.1 Runtime Overheads. Figure 8 summarizes the performance overheads of running our workloads with cuCatch's compiler instrumentation and driver support while enabling our compile-time optimizations (explained in Section 4.6). We instrument global, shared, and local memory accesses in device-side GPU code. We do not instrument host-side CPU code, as stated in Section 2.3. The workloads are ordered from left to right in increasing order of runtime overheads.

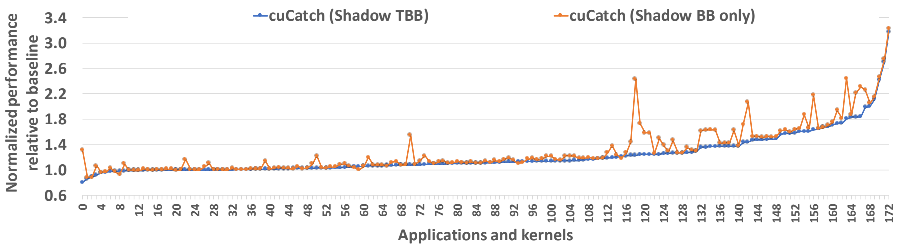


Fig. 8. Performance overheads of cuCatch with the *shadow tagged base & bounds* algorithm (Shadow TBB) and a *shadow base & bounds* only variant (Shadow BB), in which all allocations traverse the shadow map.

Most of the evaluated workloads incur negligible overheads where the geometric mean of deploying our tool is 1.19× (or 19%). To better understand the source of the overheads, we profiled

[8]We exclude tests that perform in-kernel `malloc` and CUDA Dynamic Parallelism as both features are not currently supported by our cuCatch prototype. We also exclude samples that link in pre-compiled CUDA libraries as cuCatch does not instrument binaries.

the top five slowdowns (> 2×) using nsight-compute [NVIDIA 2023b]. Our analysis shows that the `reduction` CUDA sample (#168 on the x-axis in Figure 8) suffers from memory latency stalls resulting from the introduction of bounds check instructions within a loop. Similarly, `app14` (#170) and `app23` (#172), from `amber18`, are also limited by memory access latency and register spills. The `amber18` kernels only show a 5% overhead when we limit instrumentation to global accesses. `app38` (#169) is a small `FUN3D` kernel that does not fully utilize the GPU. Hence, there is insufficient warp-level parallelism to hide the latency of the additional instructions introduced by cuCatch. When we instrument only global memory accesses, the overheads for `app38` fall to 30%. Finally, `app58` (#171) is an `RTM` kernel that shows a significant reduction in warp occupancy and increase in spills due to high register pressure even when we only instrument global memory accesses. On the other hand, we notice an unexpected performance speedup over baseline for `interval` (#4) because our instrumentation uses two additional registers than the baseline compilation, which lowers occupancy and boosts performance over the baseline by about 5%. We also observe this performance improvement when we force baseline compilation to use two extra registers.

6.2.2 Sensitivity Analysis. In addition to our main Shadow TBB configuration, we evaluated an alternate configuration that forces all metadata fetch instructions to traverse the shadow map by reducing the available upper pointer bits to zero. Doing so eliminates the benefits of directly indexing the BST using the upper pointer bits. This approach, referred to as *Shadow BB-only* in Figure 8, incurs a geometric mean of 1.25× (or 25%) runtime overheads with a maximum of 3.22×.

6.2.3 Memory Overheads. cuCatch has two sources of memory overheads: a compulsory memory cost and a scalable one. The compulsory memory overhead comes from maintaining the BST (a 32 MB data structure with $2^{20} \times 32$ B-aligned allocation metadata) and the first level of the shadow map (a 128 MB array with 2^{24} elements, where each element can hold a pointer to a second level shadow map table). The scalable component comes from the second-level shadow map tables which are allocated on demand (32-bit entry per each allocated 32 B memory region or 12.5%) when the number of allocations exceeds the upper pointer bit capacity.

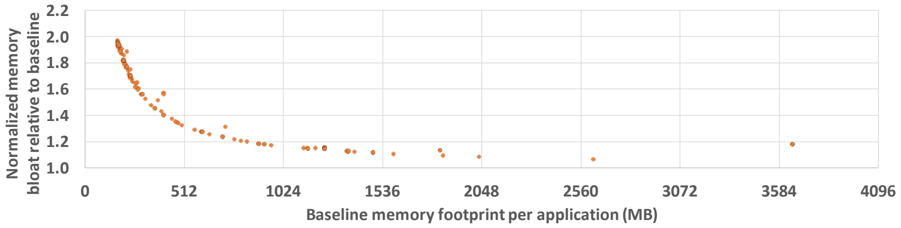


Fig. 9. Memory overheads of cuCatch (Shadow TBB) normalized to baseline execution.

Figure 9 shows the overall memory bloat of cuCatch relative to the baseline program memory footprint. When the memory footprint of the workload is small (left-hand-side of the x-axis), the memory overhead of cuCatch is comparatively large because of the compulsory cost described above. On the other hand, cuCatch's memory bloat drops significantly as the baseline memory footprint increases. For example, the overall memory overheads are less than 20% for any realistically-sized application. None of the evaluated workloads exceeds the 12 GB memory capacity of the GPU we use. Even if a memory-bound application exceeds the GPU memory capacity with cuCatch, it can still run without crashing if the metadata structures are allocated in "unified memory", which is GPU-accessible memory that migrates between the CPU and GPU. Experimenting with unified memory-based metadata structures is part of ongoing work and beyond the scope of this paper.

6.2.4 Occupancy and Divergence. Occupancy is a measure of active warps on a GPU and depends on the per-thread resource (i.e., register and shared memory) consumption. In our study, we observe a reduction in occupancy due to increase in register usage for 14 out of 176 kernels with slowdowns in 11 of the 14 kernels. Furthermore, 23 of the kernels showed an increase in spills. On the other hand, thread divergence occurs when threads within a warp execute different control flow paths at a branch instruction, resulting in in serialized execution. While cuCatch introduces conditional branch instructions, we found that the vast majority of the time, the branch conditions are warp invariant because SIMT memory operations convergently access the same memory object.

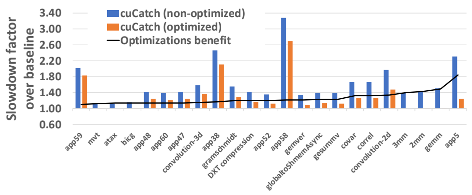


Fig. 10. cuCatch optimization effects.

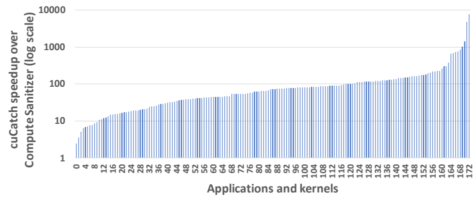


Fig. 11. cuCatch performance relative to NVIDIA's Compute Sanitizer memcheck tool.

6.2.5 cuCatch Optimizations. To quantify the impact of our proposed optimizations, Figure 10 shows a subset of the workloads where optimizations were most effective. For example, bounds check optimizations improve `app5`'s performance by 80%. Since `app5` has a large number of strided shared memory accesses, removing bounds check instructions significantly reduces the total number of instructions executed. This enables the compiler to freely issue memory accesses back-to-back for higher instruction level parallelism. On average our optimizations have a 4% performance improvement over the unoptimized version across all evaluated workloads.

6.2.6 Comparison with the State-of-the-art Error Detection Tool. Figure 11 shows the relative performance between cuCatch and NVIDIA's Compute Sanitizer memcheck tool. cuCatch is significantly faster (63× on average) primarily due to technology differences between compiler instrumentation and dynamic binary instrumentation [Villa et al. 2019]. Additionally, Compute Sanitizer detects API and misaligned memory access errors which further adds to its overall slowdowns.

7 DISCUSSION

We now discuss cuCatch limitations and how we plan to address them.

Custom memory allocators. CUDA programmers tend to wrap device allocation APIs (e.g., `cudaMalloc`) with their own memory allocators for many reasons. Some of these reasons include achieving consistent allocation time (as `cudaMalloc` timing varies even for the same allocation size) and avoiding synchronizing all streams upon memory allocations. Using a custom allocator will thwart our tool's bounds checking as we will be unaware of the base addresses of internal allocations. As many GPU libraries that use custom allocators rely on a well-defined interface for managing memory (e.g., `DeviceAllocate`/`DeviceFree` in CUB [NVIDIA 2022b] and Thrust [NVIDIA 2022g] and `do_allocate`/`do_deallocate` in the RAPIDS memory manager [NVIDIA 2022f]), one potential solution is to capture the calls to the custom memory allocator APIs similarly to how we intercept calls to the default CUDA memory allocator, and update our metadata structures accordingly.

Address translation services. Modern NVIDIA GPUs add support for a new feature called Address Translation Service (ATS) [Sakharnykh 2019]. This feature allows the programmer to transparently transfer data between the CPU and GPU without using explicit APIs such as cudaMemcpy. For example, a CUDA programmer can allocate a buffer in CPU memory using a call to the CPU-side malloc and later pass this buffer as an argument to a GPU kernel. The underlying hardware will then automatically migrate the referenced memory page from the CPU memory to GPU memory. In order for cuCatch to detect memory safety errors in ATS-migrated buffers, we will need to intercept calls to CPU-side allocators and include their bounds in a BST so we can verify the bounds of those buffers when they are accessed from the GPU-side code. We leave this extension to future work.

Backend instrumentation. In this work, we opt to implement cuCatch as an extension to the compiler backend. This provides multiple benefits. First, we can handle a variety of programming languages that target NVIDIA GPUs, including CUDA, OpenACC, DirectX, and Vulkan. Second, our tool takes advantage of the multiple optimization passes that run in the compiler backend. Inserting our instrumentation code early in the compilation pipeline (e.g., compiler front-end) may render many of the backend optimizations ineffective and degrade performance. On the other hand, the main disadvantage of working in the backend is that some semantic front-end information is lost (e.g., base pointer information and precise size information of statically-allocated local buffers). Such high-level semantics could help cuCatch achieve higher error detection coverage for global and local memory, as discussed in Section 5.2.3. One potential solution to address this problem is to explicitly propagate additional metadata from the compiler front-end to our backend.

Base pointer analysis limitations. Our compile-time *base pointer analysis* might fail to identify the true base address of an object if it comes across a destructive redefinition. Because our static analysis is intraprocedural, function parameters and return values are treated as destructive redefinitions.If the base pointer analysis fails to identify the true base pointer, traversing the shadow map might lead to fetching the base and size of an unrelated object. The same issue might occur if a pointer is stored to memory (after a pointer arithmetic operation, p=p+offset), and is then loaded from memory (before a memory access operation, *p) because our analysis ends at a memory access since this is a destructive redefinition. To address this limitation, cuCatch relies on the random 4-bit tag to probabilistically catch such memory safety errors, as the 4-bit tag from the upper pointer bits (e.g., *p) is unlikely to match the tag which is fetched from the BST entry pointed-to by p+offset.

Supporting off-by-one pointers. For an array A[N], the C standard allows the generation of pointers to (1) any element within the array (i.e., A[0], A[1], ..., A[N-1]) and (2) one element past the end of the array (i.e., A[N]), which we refer to as an off-by-one pointer. Off-by-one pointers are problematic for allocations that traverse the shadow map as the off-by-one pointer of an allocation A might end-up pointing to the first shadow map entry of an adjacent allocation B, leading to incorrect metadata retrieval. In order to smoothly support off-by-one pointers, cuCatch adds four padding bytes for each memory allocation to force the creation of an additional shadow map entry between adjacent objects. This way an off-by-one pointer that corresponds to allocation A will always retrieve the BST-entry index of its own allocation when traversing the shadow map.

JIT-code and pre-compiled libraries. We note that since its inception, CUDA has featured just-in-time compilation capabilities. By default, when a developer compiles an application, the compiler generates a binary in which the program's PTX representation is embedded. The CUDA driver can just-in-time compile the program's PTX to generate optimized machine code that runs on the target device. We can leverage this feature to instrument any application distributed with embedded PTX. However, some libraries do not contain PTX, but instead are comprised of executable machine code. Therefore, a complete solution for handling all CUDA applications and libraries requires a binary-instrumentation fallback to supplement a more efficient compiler-based instrumentation.

Table 3. Comparison with prior memory safety works on CPUs and GPUs.

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.

8 RELATED WORK

In this section, we review existing memory safety work on both CPUs and GPUs. Table 3 compares the different proposals based on the target platform, the instrumentation technology (whether the proposal is realized completely in software, requires hardware changes, or both), the error detection coverage, the metadata storage overhead, and the main sources of memory and performance overheads. We refrain from using absolute memory/performance numbers in the table as the compared proposals were evaluated on different platforms using different workloads.

8.1 CPU Solutions

8.1.1 Hardware-assisted Techniques. There exists a rich literature on addressing the C and C++ memory unsafety problem on CPUs. Prior proposals adopted different memory safety algorithms as discussed in Section 2.2.2. For example, REST [Sinha and Sethumadhavan 2018] and Califorms [Sasaki et al. 2019] implemented tripwires or memory blocklisting. REST [Sinha and Sethumadhavan 2018] stored a predetermined 8–64 B random token in the memory to be invalidated and detected illegal accesses to it by comparing cache lines with the token when they are fetched. To achieve fine-grained spatial safety detection, Califorms [Sasaki et al. 2019] inserts smaller sized tokens (1-7 B) between fields of the same allocation and changed how data is stored in cache lines for inlining the metadata within the program data without causing severe memory fragmentation. Memory tagging implementations include SPARC's Application Data Integrity (ADI) [Oracle 2015] which associates 4-bit tags with each 64 B cache line and ARM's Memory Tagging Extension (MTE) [ARM 2019], which uses 4-bit tags per each 16 B of memory.

Finally, hardware-assisted proposals that adopt the base & bounds algorithm include CHERI [Watson et al. 2015], which increases the pointer size to include the bounds information of the pointed-to allocation, Intel's MPX [Oleksenko et al. 2018] and CHEx86 [Sharifi and Venkat 2020], which store the bounds information in page-table-like format that is indexed using the pointer location in memory, and No-FAT [Tarek Ibn Ziad et al. 2021], which implicitly derives the bounds information of an allocation from its location in memory by taking advantage of binning memory allocators. There is also AOS [Kim et al. 2020], which uses the upper pointer bits to store a Pointer-Authentication-Code, which acts as a key to a hashed metadata table that holds the allocation base and size. These proposals provide temporal safety by using virtual memory quarantining, randomization of tags/colors (e.g., ARM's MTE [ARM 2019]), or automatic garbage collectors (e.g., CHERIVoke [Xia et al. 2019]).

One common theme between the aforementioned techniques is that they all require special hardware support to reduce the memory and runtime costs of memory safety. As a result, they cannot be directly used on legacy/unmodified hardware.

8.1.2 Software Tools. Existing CPU-only memory safety solutions incur high memory and runtime overheads. For example, Valgrind [Nethercote and Seward 2007] maintains a configurable number of metadata bits per each program byte/word. This metadata is stored in a shadow memory and is accessed upon each program load or store to verify whether the memory access is valid or not. To maintain perfect compatibility with legacy binaries and hardware, Valgrind uses dynamic binary instrumentation (DBI), which intercepts each program instruction before being executed on hardware and inserts the tool-specific instructions before or after it. Unfortunately, DBI results in significant (orders-of-magnitude) performance overheads.

Other CPU-based tools take advantage of compiler support for instrumenting programs to avoid the high cost of DBI. For example, Google's Address Sanitizer [Serebryany et al. 2012] uses shadow memory to detect OOB errors, uninitialized reads, and certain temporal memory safety violations in addition to SoftBound [Nagarakatte et al. 2009], which was discussed in Section 3.3.

8.2 GPU Solutions

8.2.1 Software Tools. Current GPUs have few protections against memory safety-based vulnerabilities. To detect memory safety errors on GPUs, NVIDIA's Compute Sanitizer memcheck tool can be used [NVIDIA 2022a]. Since Compute Sanitizer uses DBI, it incurs large performance overheads. Other tools, such as clARMOR [Erb et al. 2017] and GMOD [Di et al. 2021, 2018], surround buffers with an area of known values called canaries. They detect buffer overflows by periodically verifying that the canary value has not been altered. Unfortunately, canaries cannot detect OOB reads by construction. Moreover, canaries are ineffective against non-adjacent OOB violations.

8.2.2 Hardware-assisted Techniques. GPUShield [Lee et al. 2022] is a recent GPU-focused memory safety solution that proposes hardware modifications to catch spatial safety violations. As discussed in Section 3.3, GPUShield solely relies on upper pointer bits to store the bounds table entry, which precludes protecting an arbitrarily large number of allocations, especially on emerging 57-bit systems. Further, the design choice of creating a per-kernel bounds table limits GPUShield's ability to capture temporal safety violations as it lacks the whole-execution metadata view.

9 CONCLUSION

With the rapidly-growing adoption of GPUs in various applications, ranging from artificial intelligence and medical imaging to physics simulation and blockchain, it is crucial to develop efficient tools that help GPU developers uncover various errors in their applications. This paper presents cuCatch, a software-based tool for efficiently detecting memory safety errors in CUDA applications. We implemented cuCatch as an extension to NVIDIA's backend compiler and driver and showed that it incurs low runtime overheads compared to state-of-the-art debugging tools. Moreover, we tested the error detection coverage of cuCatch using multiple benchmarks with memory safety errors. We show that cuCatch provides the highest error detection coverage compared to state-of-the-art memory safety tools for GPUs due to its novel underlying memory safety algorithm and optimized compiler implementation. The low runtime overheads and high error detection coverage allow cuCatch to stand as an efficient debugging tool during development and testing.

REFERENCES

Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 2007. *Compilers: principles, techniques, and tools*. Vol. 2. Addison-wesley Reading.

ARM. 2019. Memory Tagging Extension: Enhancing memory safety through architecture. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/enhancing-memory-safety. [Online].

Joe Bialek, Ken Johnson, Matt Miller, and Tony Chen. 2020. Security analysis of memory tagging. https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20memory%20tagging.pdf

Bang Di, Jianhua Sun, and Hao Chen. 2016. A study of overflow vulnerabilities on GPUs. In *NPC '16: Proceedings of the 13th International Conference on Network and Parallel Computing*. Xi'an, China, 103–115. https://doi.org/10.1007/978-3-319-47099-3_9

Bang Di, Jianhua Sun, Hao Chen, and Dong Li. 2021. Efficient buffer overflow detection on GPU. *IEEE Transactions on Parallel Distributed Systems* 32, 5 (2021), 1161–1177. https://doi.org/10.1109/TPDS.2020.3042965

Bang Di, Jianhua Sun, Dong Li, Hao Chen, and Zhe Quan. 2018. GMOD: a dynamic GPU memory overflow detector. In *PACT '18:Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. Limassol, Cyprus, 20:1–20:13. https://doi.org/10.1145/3243176.3243194

Christopher Erb, Mike Collins, and Joseph L. Greathouse. 2017. Dynamic buffer overflow detection for GPGPUs. In *CGO '17: Proceedings of the 2017 IEEE/ACM International Symposium on Code Generation and Optimization*. Austin, TX, USA, 61–73. https://doi.org/10.1109/CGO.2017.7863729

Christopher Erb and Joseph L. Greathouse. 2018. clARMOR: A dynamic buffer overflow detector for OpenCL kernels. In *IWOCL '18: Proceedings of the International Workshop on OpenCL*. Oxford, United Kingdom, Article 15, 2 pages. https://doi.org/10.1145/3204919.3204934

Google. 2019. 0day In the Wild. https://googleprojectzero.blogspot.com/p/0day.html

Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *InPar '12: Proceedings of the 2012 Innovative Parallel Computing*. San Jose, CA, USA, 1–10. https://doi.org/10.1109/InPar.2012.6339595

Mark Harris. 2013. Using shared memory in CUDA. https://developer.nvidia.com/blog/using-shared-memory-cuda-cc

Aditya K. Kamath and Arkaprava Basu. 2021. IGUARD: In-GPU advanced race detection. In *SOSP '21: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. Virtual Event, Germany, 49–65. https://doi.org/10.1145/3477132.3483545

Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. 2020. Hardware-based always-on heap memory safety. In *MICRO-53: Proceedings of the 53nd Annual IEEE/ACM International Symposium on Microarchitecture*. Global Online Event, 1153–1166. https://doi.org/10.1109/MICRO50266.2020.00095

Jaewon Lee, Yonghae Kim, Jiashen Cao, Euna Kim, Jaekyu Lee, and Hyesoon Kim. 2022. Securing GPU via region-based bounds checking. In *ISCA '22: Proceedings of the 49th Annual International Symposium on Computer Architecture*. New York, NY, USA, 27–41. https://doi.org/10.1145/3470496.3527420

Andrea Miele. 2016. Buffer overflow vulnerabilities in CUDA: a preliminary analysis. *Journal of Computer Virology and Hacking Techniques* 12, 2 (2016), 113–120. https://doi.org/10.1007/s11416-015-0251-1

Matt Miller. 2019. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. In *BlueHat*. https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf

Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for C. In *PLDI '09: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Dublin, Ireland, 245–258. https://doi.org/10.1145/1542476.1542504

Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. San Diego, CA, USA, 89–100. https://doi.org/10.1145/1250734.1250746

NVIDIA. 2022a. Compute Sanitizer v2022.2.1. https://docs.nvidia.com/cuda/sanitizer-docs/pdf/ComputeSanitizer.pdf

NVIDIA. 2022b. CUB: Cooperative primitives for CUDA C++. https://github.com/NVIDIA/cub

NVIDIA. 2022c. CUDA-MEMCHECK v11.8.0. https://docs.nvidia.com/cuda/pdf/CUDA_Memcheck.pdf

NVIDIA. 2022d. CUDA Samples. https://github.com/NVIDIA/cuda-samples

NVIDIA. 2022e. Parallel Thread Execution ISA v11.8.0. https://docs.nvidia.com/cuda/pdf/ptx_isa_7.8.pdf

NVIDIA. 2022f. RMM: RAPIDS Memory Manager. https://github.com/rapidsai/rmm

NVIDIA. 2022g. Thrust: The C++ Parallel Algorithms Library. https://github.com/NVIDIA/thrust

NVIDIA. 2023a. CUDA C Programming Guide, Section 10.34: Dynamic Global Memory Allocation and Operations. NVIDIA Developer Zone. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf Release 12.0.

NVIDIA. 2023b. Nsight Compute. https://docs.nvidia.com/nsight-compute/NsightCompute/index.html

Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel MPX explained: A cross-layer analysis of the intel MPX system stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 2 (2018), 28. https://doi.org/10.1145/3224423

Oracle. 2015. Hardware-assisted checking using Silicon Secured Memory (SSM). https://docs.oracle.com/cd/E37069_01/html/E37085/gphwb.html

Sang-Ok Park, Ohmin Kwon, Yonggon Kim, Sang Kil Cha, and Hyunsoo Yoon. 2021. Mind control attack: Undermining deep learning with GPU memory exploitation. *Computers & Security* 102 (2021), 102–115. https://doi.org/10.1016/j.cose.2020.102115

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: A general purpose ray tracing engine. *ACM Trans. Graph.* 29, 4, Article 66 (jul 2010), 13 pages. https://doi.org/10.1145/1778765.1778803

Yuanfeng Peng, Vinod Grover, and Joseph Devietti. 2018. CURD: A dynamic CUDA race detector. In *PLDI '18:Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Philadelphia, PA, USA, 390–403. https://doi.org/10.1145/3192366.3192368

Nikolay Sakharnykh. 2019. Memory management on modern GPU architectures. In *GTC '19: the 10th annual GPU Technology Conference*. San Jose, CA, USA.

Hiroshi Sasaki, Miguel A. Arroyo, Mohamed Tarek Ibn Ziad, Koustubha Bhat, Kanad Sinha, and Simha Sethumadhavan. 2019. Practical byte-granular memory blacklisting using Califorms. In *MICRO-52: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Columbus, OH, USA, 558–571. https://doi.org/10.1145/3352460.3358299

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *ATC '12: Proceedings of the 2012 USENIX Annual Technical Conference*. Boston, MA, USA.

Rasool Sharifi and Ashish Venkat. 2020. CHEx86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities. In *ISCA '20: Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*. Valencia, Spain, 762–775. https://doi.org/10.1109/ISCA45697.2020.00068

Kanad Sinha and Simha Sethumadhavan. 2018. Practical memory safety with REST. In *ISCA '18: Proceedings of the 45th Annual International Symposium on Computer Architecture*. Los Angeles, CA, USA, 600–611. https://doi.org/10.1109/ISCA.2018.00056

Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for security. In *SP '19: Proceedings of the IEEE Symposium on Security and Privacy*. San Francisco, CA, USA, 1275–1295. https://doi.org/10.1109/SP.2019.00010

Arthy Sundaram, Jaydeep Marathe, Mike Murphy, Nikhil Gupta, Xiaohua Zhang, and Thibaut Lutz. 2021. Programming Efficiently with the NVIDIA CUDA 11.3 Compiler Toolchain: Preview support for alloca. https://developer.nvidia.com/blog/programming-efficiently-with-the-cuda-11-3-compiler-toolchain/

Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal war in memory. In *SP '13: Proceedings of the 2013 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA, 48–62. https://doi.org/10.1109/SP.2013.13

Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, Ryan Piersma, and Simha Sethumadhavan. 2021. No-FAT: Architectural support for low overhead memory safety checks. In *ISCA-48: Proceedings of the 48th Annual International Symposium on Computer Architecture*. Worldwide Event, 916–929. https://doi.org/10.1109/ISCA52012.2021.00076

Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. 2012. Memory errors: The past, the present, and the future. In *Research in Attacks, Intrusions, and Defenses*, Davide Balzarotti, Salvatore J. Stolfo, and Marco Cova (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 86–106. https://doi.org/10.1007/978-3-642-33338-5_5

Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A dynamic binary instrumentation framework for NVIDIA GPUs. In *MICRO-52: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Columbus, OH, USA, 372—-383. https://doi.org/10.1145/3352460.3358307

Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert M. Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. 2015. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *SP '15: Proceedings of the 2015 IEEE Symposium on Security and Privacy*. San Jose, CA, USA, 20–37.

Mingyuan Wu, Yicheng Ouyang, Husheng Zhou, Lingming Zhang, Cong Liu, and Yuqun Zhang. 2020. Simulee: detecting CUDA synchronization bugs via memory-access modeling. In *ICSE '20: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Seoul, South Korea, 937–948. https://doi.org/10.1145/3377811.3380358

Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and et al. 2019. CHERIvoke: Characterising pointer revocation using CHERI capabilities for temporal memory safety. In *MICRO-52: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Columbus, OH, USA, 545–557. https://doi.org/10.1145/3352460.3358288

Received 2022-11-10; accepted 2023-03-31