M2K: Making the Model-Kernel Interface Explicit for Reliable CUDA Kernel Verification

Mengting He
Penn State
USA

Shihao Xia
Penn State
USA

Haomin Jia
SKLP, ICT, CAS
China

Wenfei Wu
Peking University
China

Linhai Song
SKLP, ICT, CAS
China

Abstract

Large language model (LLM) inference systems rely on CUDA kernels for core GPU computations, yet the interface between models and kernels is implicit and poorly specified. Models and kernels evolve independently and often make incompatible assumptions about tensor shapes and input sizes, leading to subtle memory bugs in CUDA kernels. These bugs can crash inference services, corrupt model weights, or be exploited by remote adversaries. Existing techniques either incur prohibitive runtime overhead, require specialized hardware, or fail to handle dynamic tensor shapes and variable kernel launch configurations, leaving the CUDA memory bugs largely unaddressed.

This paper presents M2K, a fully automated framework that makes the model-kernel interface explicit and leverages it to detect memory bugs in CUDA kernels used in LLM inference systems. M2K consists of two components. HFProbe traces model execution without GPU hardware, classifies kernel arguments into model-fixed and user-variable, and emits symbolic constraints that capture the interface. cuKLEE then performs symbolic execution on CUDA kernels to pinpoint memory bugs under the interface constraints, modeling tensors as disjoint memory regions and treating thread identifiers symbolically to scale to thousands of threads. In the evaluation, M2K discovers 181 previously unknown bugs in real LLM inference systems, while producing only nine false positives, demonstrating its effectiveness.

CCS Concepts: • **Software and its engineering → Empirical software validation**; *Software infrastructure.*

ACM Reference Format:
Mengting He, Shihao Xia, Haomin Jia, Wenfei Wu, and Linhai Song. 2026. M2K: Making the Model-Kernel Interface Explicit for Reliable CUDA Kernel Verification. In *ACM SIGOPS 32nd Symposium on Operating Systems Principles (SOSP '26), September 29-October 02, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3830418.3843908

SOSP '26, Prague, Czech Republic

ACM ISBN 979-8-4007-2585-2/2026/09
https://doi.org/10.1145/3830418.3843908

1 Introduction

The rapid adoption of large language models (LLMs) has driven an explosion of inference workloads across chatbots [1, 2, 16], search [38, 86], recommendation [5, 75, 81], copilots [12, 33, 85], scientific computing [7, 39, 74], and countless other areas [8, 9, 79]. These services now operate at scale and serve real users, making LLM inference systems a critical part of modern computing infrastructure.

LLM inference systems rely on a layered software architecture. Model developers publish Python model files and weights (via Hugging Face [24]), which inference frameworks (*e.g.*, vLLM [35], SGLang [82], Transformers [25]) load and execute. These frameworks manage inference requests and map high-level tensor operations in Python model code to host-side wrapper code. The wrapper code prepares CUDA kernel inputs and configures kernel launches. At the lowest layer, CUDA kernels perform the core linear algebra computation on the GPU. This separation enables independent development at each layer and accelerates system evolution.

However, the layered architecture introduces a fragile, implicit interface between models and CUDA kernels. The correctness of the system depends on assumptions made independently by the two sides, yet these assumptions are neither explicitly specified nor consistently enforced. The mismatch manifests in two ways. First, kernels may contain outright implementation errors in the intricate performance optimizations (*e.g.*, vectorization, tiling), causing them to produce correct results only for specific tensor shapes, while silently performing out-of-bounds accesses for others. Second, kernels may impose implicit limits on input sizes (*e.g.*, due to integer types used for index arithmetic or fixed-size internal buffers) that were safe when the kernel was written but are increasingly violated as models grow larger [79] and inference sequences become longer [17, 71]. Both classes of mismatch lead to kernel memory bugs (*e.g.*, buffer overflows, integer overflows) that are difficult to catch, because they surface only under specific combinations of model architecture, runtime input, and kernel implementation.

These kernel memory bugs carry severe consequences. Out-of-bounds accesses and data races trigger "illegal memory access" errors on the GPU, potentially crashing the entire inference service [60]. More critically, recent work has shown that memory bugs on GPUs can be exploited to modify arbitrary GPU memory or achieve arbitrary code execution [19, 27, 52]. Since inference systems are exposed to the Internet and process user-controlled prompts, attackers can

choose sequence length and influence server-side batching. As a result, exploitable memory bugs in CUDA kernels form a real attack surface that may allow attackers to compromise the host system or affect other service users, as demonstrated by our PoC exploits [28, 29].

Existing techniques for detecting memory bugs in CUDA kernels fall short in this setting. Dynamic methods track metadata for GPU memory buffers and validate pointer dereferences at runtime [20–22, 37, 44, 54, 55, 69, 70, 77, 83, 84, 87]. However, they either incur high overhead or require hardware extensions not yet widely available. More fundamentally, they analyze kernels in isolation from the models that invoke them, leading to both false positives and false negatives. Static tools [6, 11, 40–42, 45, 46, 49, 62, 72] cannot handle runtime-determined buffer sizes or the variability of kernel launch configurations, both of which are common in inference systems. Techniques for testing machine learning frameworks [48, 58] mainly target the framework layer and cannot pinpoint CUDA kernel bugs. In short, no existing technique bridges the gap between model-level usage and kernel-level implementation to provide comprehensive memory-bug detection for CUDA kernels.

We make a key observation: the implicit, unspecified interface between models and CUDA kernels is not merely a source of kernel memory bugs—it is an *architectural weakness* of current LLM inference stacks. Models and kernels evolve on independent release cycles, maintained by different teams with different assumptions, yet no contract governs what one side may assume about the other. Effectively detecting memory-safety bugs in CUDA kernels therefore requires making this interface explicit: we must understand how each model invokes its kernels and use that information to systematically verify two properties—whether model usage respects kernel limitations, and whether kernel implementations are safe under all inputs the model can produce.

Realizing this idea raises two key challenges. First, *how to automatically extract representative kernel usage patterns from a model.* Kernel arguments are jointly determined by model architecture, runtime configuration, and user-controlled inputs (*e.g.*, sequence length, batch size). A practical solution must effectively characterize this combinatorial space of possible invocations to provide actionable guidance for CUDA kernel analysis. Second, *how to accurately and scalably verify CUDA kernels under the resulting usage conditions.* Kernels operate on tensors whose shapes are determined at runtime, a setting that existing static analyses handle poorly. Moreover, kernels execute across thousands of parallel threads, and naïvely enumerating those threads causes state explosion, making exhaustive verification infeasible.

To address these challenges, we design M2K, a fully automated analysis framework that detects memory-safety issues in CUDA kernels as they are used by a given LLM model. M2K targets four types of bugs in CUDA kernels: buffer overflows, integer overflows, data races, and null-pointer

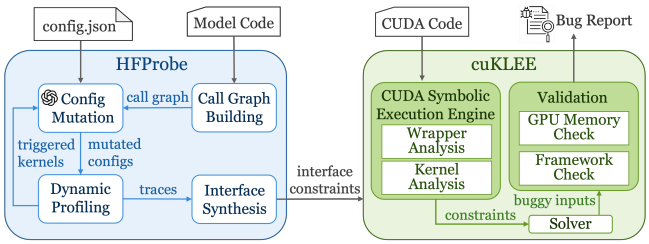

Figure 1. An overview of M2K.

dereferences, and supports the standard Hugging Face model format, covering the vast majority of publicly available LLMs. As shown in Figure 1, M2K consists of two major components: HFProbe and cuKLEE.

HFProbe "executes" each model without requiring GPU hardware. It profiles kernel invocations by simulating execution without running the kernels, thereby achieving high efficiency. It classifies kernel arguments into those fixed by the model architecture (*e.g.*, hidden dimension) and those varying with user input (*e.g.*, sequence length, batch size). HFProbe translates this classification into symbolic constraints that make the model-kernel interface explicit and passes them to cuKLEE for bug detection. In addition, HFProbe statically identifies all CUDA kernels reachable from a given model and automatically mutates the model's configurations to trigger more kernels, ensuring comprehensive analysis.

cuKLEE performs symbolic execution on CUDA kernels with the interface constraints from HFProbe. To handle input tensors whose memory layouts are determined at runtime, cuKLEE models each tensor as a distinct, widely separated memory region. This abstraction simplifies pointer–tensor association and makes bounds checking more tractable. cuKLEE introduces symbolic variables to represent each tensor's base address, element count, dimensions, and other properties. With these variables, M2K can handle more than 100 tensor API methods by adding the appropriate constraints to its execution states. It also models CUDA thread identifiers symbolically, allowing a single pass to capture both shared and thread-specific computations across thousands of threads. Finally, cuKLEE includes a memory usage model that accounts for the input token count, model size, and available GPU memory. It uses the model to determine whether a detected memory bug can actually occur on a target GPU, ensuring that reported bugs are actionable in practice.

We evaluate M2K on CUDA kernels and models from three sources: all text-generation models in vLLM [35] and the related kernels, all Hugging Face models with custom kernels, and models and kernels from four recent research publications [32, 50, 61, 76]. In total, M2K identifies 181 memory bugs, including 162 integer overflows [1] and 19 out-of-bounds accesses, with only nine false positives, demonstrating its effectiveness in detecting memory safety issues for

[1]We suspect that integer overflows are prevalent in our detection results because they are more likely to cause silent data corruption than crashes.

```python
class Qwen2ForCausalLM(nn.Module):
  def __init__(self, config):
    decoder_layer_type = Qwen2DecoderLayer
    self.layers = ...
  def forward(self, input_ids, positions):
    hidden_states, residual = ...
    for layer in self.layers:
      hidden_states, residual = layer(positions, hidden_states,
      residual)
    return hidden_states
class Qwen2DecoderLayer(nn.Module):
  def __init__(self, config):
    self.fp8_linear = Fp8LinearOp()
  def forward(self, positions, x, residual):
    out_dtype = torch.get_default_dtype()
    hidden_states = self.fp8_linear.apply(x, out_dtype)
    ...
    return hidden_states, residual
class Fp8LinearOp:
  def apply(self, input, out_dtype):
    in_2d = input.view(-1, input.shape[-1])
    qinput = torch.empty(in_2d.shape, dtype=out_dtype)
    scale = torch.zeros(1, dtype=torch.float32)
    torch.ops._C.dynamic_scaled_fp8_quant(qinput, in_2d, scale)
    return w8a8_scaled_mm_func(...)
```

(a) The simplified Python model code of Qwen2ForCausalLM.

```
TORCH_LIBRARY_EXPAND(TORCH_EXTENSION_NAME, ops) {
  ops.def("dynamic_scaled_fp8_quant(Tensor! result, Tensor input,
  Tensor! scale) -> ()");
  ops.impl("dynamic_scaled_fp8_quant", torch::kCUDA,
  &dynamic_scaled_fp8_quant);
}
```

(b) The registration of a CUDA kernel in Python.

```
void dynamic_scaled_fp8_quant(torch::Tensor& out, torch::Tensor
const& input, torch::Tensor& scale) {
  int64_t num_tokens = input.numel() / input.size(-1);
  int64_t num_elems = input.numel();
  dim3 grid(num_tokens);
  dim3 block(1024);
  using bf16 = at::BFloat16; using fp8 = at::Float8_e4m3fn;
  vllm::scaled_fp8_quant_kernel<bf16, fp8>
    <<<grid, block, 0>>>(out.data_ptr<fp8>(),
    input.data_ptr<bf16>(), scale.data_ptr<float>(), num_elems);
}
```

(c) Wrapper function.

```
template <typename scalar_t, typename fp8_type>
__global__ void scaled_fp8_quant_kernel(fp8_type* __restrict__
out, const scalar_t* __restrict__ input, const float*
__restrict__ scale, int64_t num_elems) {
  int tid = blockDim.x * blockIdx.x + threadIdx.x; // overflow
  const float inverted_scale = 1.0f / (*scale);
  int step = blockDim.x * gridDim.x; // overflow

  using float8x4_t = q8x4_t<fp8_type>;
  auto* v_in = reinterpret_cast<vec4_t<scalar_t> const*>(input);
  auto* v_out = reinterpret_cast<float8x4_t*>(out);
  int64_t const num_vec_elems = num_elems >> 2;

  for (int64_t i = tid; i < num_vec_elems; i += step) {
    vec4_t<scalar_t> in_vec = v_in[i]; // OOB
    v_out[i] = scaled_fp8_conversion<fp8_type> // OOB
      (in_vec, inverted_scale);
  }
  ...
}
```

(d) CUDA kernel. *(Thread-dependent computations are in yellow.)*

Figure 2. A CUDA kernel example, its wrapper function, its registration to vLLM, and a Hugging Face model using the kernel.

CUDA kernels. We have reported all 26 bugs detected in vLLM. So far, eight bugs have been fixed in subsequent releases and additional four bugs have been confirmed by the programmers, underscoring M2K's practicality. We further compare M2K with Honeycomb [49], GKLEE [41], and ESBMC-GPU [62] on 20 known vLLM CUDA kernel bugs. M2K successfully detects 15 of these bugs, far more than the baseline techniques, highlighting its superior bug coverage and clear improvement over existing techniques. An ablation study shows that both HFProbe and cuKLEE are critical for M2K's effectiveness. In sum, we make the following contributions:

- We identify the implicit model-kernel interface as one source of memory bugs in CUDA kernels used in LLM inference systems and argue that making this interface explicit is the key to effective bug detection.
- We develop HFProbe, a dynamic model profiler that traces how models invoke CUDA kernels and infers model-kernel interfaces, without GPU hardware.
- We design cuKLEE, a symbolic execution engine specialized for CUDA kernels that supports tensors with dynamic shapes and scales to thousands of threads.
- By integrating HFProbe and cuKLEE, we build M2K, a fully automated framework for detecting CUDA memory bugs in LLM inference systems and demonstrate its effectiveness via thorough experiments.

M2K is publicly available at https://github.com/system-pclub/M2K.

2 Background

This section provides background for the project, including Hugging Face models, the CUDA programming language, and CUDA kernel memory bugs in LLM inference systems.

2.1 Hugging Face Models

Hugging Face is the largest platform for sharing pre-trained models [24], offering both a model marketplace and libraries that facilitate model integration into LLM inference frameworks (*e.g.*, vLLM [35], SGLang [82], Transformers [25]). Today, it hosts millions of models, including major open-source foundation models (*e.g.*, Qwen, LLaMA) and their variants, making it a central hub for the AI community.

A model released on Hugging Face typically contains four types of files. The *config.json* file defines the model's architecture, including layer number, hidden size, activation functions, and other structural parameters. The model-weight files store the trained parameters. The Python code files define classes that extend `nn.Module` and implement the model architecture. One top-level class assembles all model components, with its `forward()` method as the entry of the model's whole forward pass. For example, Figure 2a shows simplified model code for Qwen2ForCausalLM, and `forward()` on line 5 begins the model's inference process. Finally, documentation files (*e.g.*, README.md) provide instructions, usage examples, and licensing information.

2.2 The CUDA Programming Language

CUDA is the dominant programming platform for exploiting GPU computational power [15], and most GPU kernels in inference systems are written in CUDA C++.

CUDA source files typically contain two types of code: host wrapper functions and GPU kernel functions. Host wrappers run on the CPU and are exposed to Python code in inference systems via registration or binding macros [30, 64] (*e.g.*, Figure 2b). They process input tensors from Python code into kernel arguments, configure launch parameters (*e.g.*, number of thread blocks, threads per block, shared-memory size), and call the kernels. CUDA kernels run on the GPU. They define the computation performed by each parallel thread. Under CUDA's execution model, a kernel runs across many threads organized into blocks and grids. Each thread uses built-in identifiers (*e.g.*, `blockIdx.x`, `threadIdx.x`) to determine its location in this hierarchy and the portion of input data it processes.

Function `dynamic_scaled_fp8_quant()` in Figure 2c is a wrapper function. It first computes the token count and feature-element count in tensor `input` on lines 2–3. It then sets the number of blocks to the token count on line 4 and the number of threads per block to 1024 on line 5. Finally, it launches kernel function `scaled_fp8_quant_kernel()` with pointers to the underlying memory of the three input tensors and the feature-element count on lines 7–8.

Figure 2d shows the CUDA kernel, which converts high-precision element values in `input` into FP8 using the scaling factor `scale`. To improve efficiency, the kernel applies vectorization. On lines 8–9, the pointers `input` and `out` are cast to vectorized types so each access processes four elements. Line 10 computes the number of such accesses by dividing the element count by four. The loop on lines 12–16 distributes work across threads using a grid-stride pattern. Each thread starts from its global index (`blockDim.x * blockIdx.x + threadIdx.x`, line 3) and moves with a stride equal to the total number of threads (`blockDim.x * gridDim.x`, line 5). In each loop iteration, a thread loads four input elements on line 13, converts them to FP8 using the reciprocal of `scale`, and writes the results back to the output buffer on line 14.

2.3 Kernel Memory Bugs in Inference Systems

CUDA kernels in LLM inference systems are prone to memory bugs. Kernels perform index arithmetic over large multi-dimensional tensors across thousands of threads; even small indexing mistakes can cause out-of-bounds accesses or data races. Performance optimizations (*e.g.*, vectorization, tiling) further complicate this arithmetic. Moreover, because models and kernels are developed independently with no formal interface contract, a kernel that is correct for one model may silently produce memory errors for another.

Table 1. Summary of 20 kernel memory bugs in vLLM.

Dimension	Breakdown	Count
How discovered	User-reported (production failure)	13
	Unit test	6
	Code review	1
Memory location	GPU global memory (tensors)	17
	GPU shared memory	3
Bug type	Integer overflow → buffer overflow	10
	Buffer overflow (other causes)	8
	Null-pointer dereference	2
Patch location	Kernel code	18
	Framework code	2

Figure 2d illustrates a representative bug rooted in the mismatch between model assumptions and kernel interface constraints. The kernel uses signed 32-bit integers for `tid` on line 3 and `step` on line 5. Both are computed as the product of the token count (`blockIdx.x` or `gridDim.x`) and 1024 (`blockDim.x`), implicitly capping the safe token count at $2^{21} - 1$. Neither the model nor the inference framework enforces this bound, as the kernel, model, and framework are developed independently with no formal interface constraints on valid input ranges. A combination of batch size BS and sequence length SL satisfying $BS \times SL > 2^{21}$ (*e.g.*, $BS = 116, SL = 18{,}079$) thus silently triggers overflow, making `tid` and `step` negative. The corrupted `tid` is then used as an index to read and write the input/output buffers, turning a silent arithmetic error into out-of-bounds memory accesses. We reproduced this bug under vLLM on an NVIDIA B300 GPU: the overflowed read on line 13 lands on an unmapped virtual address, raising an "illegal memory access" fault and crashing the inference service.

To quantify the problem, we examine vLLM's change logs and identify 20 fixed memory bugs in CUDA kernels (Table 1). Four findings stand out. First, memory bugs in CUDA kernels are common and costly. Of the 20 bugs, 13 were reported by users after causing failures in production; only six were caught by unit tests and one by code review. Second, nearly all bugs involve tensors. 17 bugs occur on input tensors in GPU global memory and three in GPU shared memory, confirming that tensor shape and layout mismatches are the dominant root cause. Third, integer overflow is the leading bug pattern. Ten of the 20 bugs are integer overflows in index computations that subsequently trigger buffer overflows, reflecting the growing risk as models and input sequences scale up. Fourth, fixing these bugs is not always limited to changing the kernel code. Two bugs were fixed by modifying the framework's kernel invocation instead. This shows that diagnosing and fixing kernel memory bugs requires reasoning across the entire software stack, rather than analyzing the CUDA kernel alone. These findings motivate M2K's design: HFProbe makes tensor-level interface constraints explicit, and cuKLEE uses symbolic execution to check index arithmetic and buffer bounds that existing testing misses.

3 Overview of M2K

M2K detects memory bugs in CUDA kernels used by LLM inference systems. As shown in Figure 1, it consists of two components: HFProbe, which constructs constraints that capture the model-kernel interface, and cuKLEE, which performs symbolic execution on CUDA kernels under these constraints to identify memory bugs.

HFProbe supports two inference backends, vLLM [35] and Transformers [25], covering most Hugging Face [24] models. Given a model, HFProbe traces its execution without launching GPU kernels, and infers whether each kernel argument is model-fixed or user-controlled and possible relationships among the arguments. Then, it synthesizes interface constraints to encode the inference results. For example, when analyzing the model in Figure 2a with the CUDA kernel in Figure 2d, HFProbe infers that tensors `out` and `input` in Figure 2c share the same shape, with their last dimensions fixed at 896 (the model's hidden size) and their first dimensions equal to the number of user-input tokens.

cuKLEE takes the interface constraints as input and checks for out-of-bounds accesses, integer overflows, data races, and null-pointer dereferences, reporting each bug's line number along with a triggering input. For example, for the CUDA kernel in Figure 2d and its wrapper in Figure 2c, cuKLEE reports that batch size 116 and sequence length 18,079 (whose product is the number of input tokens and equals the first dimension of `input` and `out` in Figure 2c) trigger integer overflows on lines 3 and 5 of Figure 2d and out-of-bounds accesses on lines 13 and 14.

M2K serves two complementary usage scenarios. Model developers can run it to verify existing kernels correctly support a new model architecture and expected token counts, while kernel developers can confirm that an optimized or rewritten kernel remains safe across all deployed models.

4 HFProbe: Dynamic Model Profiling

As illustrated in Figure 1, HFProbe operates in four stages. It first performs static analysis to identify all CUDA kernels an input model may call. It then executes the model using mocked CUDA kernels with a real model configuration, recording how the kernels are actually invoked. Next, it mutates the model configuration to exercise additional kernels identified statically but not triggered dynamically. Finally, it aggregates the collected information in a post-mortem analysis to derive constraints that summarize the interface between the model and each kernel it uses.

4.1 Static Call Graph Computation

Given a model's Python implementation (*e.g.*, Figure 2a), we build a static call graph rooted at the model's `forward()` method. The analysis is path- and field-sensitive and leverages common coding patterns of Hugging Face models (*e.g.*,

```
1   class Fp8LinearOp:
2     def apply(self, input, out_dtype):
3       in_2d = input.view(-1, input.shape[-1])
4       qinput = torch.empty(in_2d.shape, dtype=out_dtype)
5       scale = torch.zeros(1, dtype=torch.float32)
6   -     torch.ops._C.dynamic_scaled_fp8_quant(qinput, in_2d, scale)
7   +     stub_dynamic_scaled_fp8_quant(qinput, in_2d, scale)
8       return w8a8_scaled_mm_func(...)
9
10  +def stub_dynamic_scaled_fp8_quant(output, input, scale):
11  +  LOG('dynamic_scaled_fp8_quant')
12  +  LOG(retrieve_stack_info())
13  +  LOG('output', type(output), output.shape, output.dtype)
14  +  LOG('input', type(input), input.shape, input.dtype)
15  +  LOG('scale',type(scale), scale.shape, scale.dtype)
16  +  return None
```

Figure 3. A tracing stub example.

assuming a model object has no aliases within its own implementation). The analysis over-approximates the results and includes all kernels that may be invoked. These kernels guide the subsequent model configuration mutation. During dynamic profiling, HFProbe ignores any kernels that are not actually executed. Existing Python call-graph tools cannot correctly analyze Hugging Face models [31, 67, 78]. For example, PyCG does not handle `__call__`, preventing it from resolving calls to an `nn.Module` object—the standard way to invoke a layer's `forward()` method (*e.g.*, line 8 in Figure 2a).

Our analysis operates on the Abstract Syntax Tree (AST) of the input Python program. It maintains an execution context where each program object is represented as an instance, created when its class constructor is analyzed. Each instance stores its fields and their values in a dictionary. The analysis processes each instruction with respect to this context and also updates it based on the instruction's semantics.

Our analysis initializes the context from the model's `__init__()` method. It then analyzes the model's `forward()` method to construct the call graph. For each field assignment (*e.g.*, line 4 in Figure 2a), the analysis updates the left-hand instance's dictionary with either the concrete value or the right-hand-side instance. For each function call, it performs interprocedural analysis by mapping the actual-argument instances to formal parameters. If the receiver's type is known only at the base class level, all possible subclasses' methods are treated as potential callees. At each branch, the analysis inspects the context to decide feasible paths. If both paths may occur, it clones the context and analyzes each path independently. When encountering Python bindings to CUDA kernels (*e.g.*, line 23 in Figure 2a), it records that the model calls the corresponding CUDA kernels. All Python bindings and their links to CUDA kernels (*e.g.*, Figure 2b) are identified separately using regular expressions.

4.2 Profiling Backends

We modify the vLLM and Transformers inference frameworks to serve as HFProbe backends and record runtime information during inference. To speed up profiling, we disable all GPU computations, allowing models to run without GPU hardware. Overall, we make four types of changes.

First, for every registered CUDA kernel, we replace its Python binding with a tracing stub. As shown in Figure 3, the stub function records the call stack and parameter types. For tensor inputs (*e.g.*, `input`), it further records their shapes and data types. For integer-valued tensors, it also records the minimum and maximum values in the tensors to capture possible index ranges. For primitive-type parameters, the stub records their concrete values. If the original binding returns a tensor, the stub returns a zero-filled tensor with the same shape and data type.

Second, model-loading functions are modified to create tensors with correct shapes without loading actual weights.

Third, although no GPU computations are executed, we ensure the frameworks follow the same control flow as with real GPUs. This involves replacing `vllm._C` with a fake module, since it cannot be imported without a real GPU, and adjusting tensors' internal properties so the frameworks treat their data as GPU-resident.

Fourth, we optimize vLLM's KV-cache management to initialize tensors with minimal memory and expand them to the required shapes without more memory allocation. Extra memory is allocated only when a tensor receives writes beyond its initial space. This lazy allocation strategy eliminates pre-allocated memory and reduces memory pressure.

4.3 Model Configuration Mutation

For each input model, we mutate two types of execution configuration parameters to trigger CUDA kernels identified statically but not exercised during earlier profiling.

First, several parameters in the model's *config.json* govern which CUDA kernels are invoked during the forward pass. For example, changing parameter `quantization_config` selects a different quantization kernel, while reducing `n_routed_experts` from 256 to 128 alters the padding kernel used for token routing from `sgl_moe_align_block_size()` to `moe_align_block_size()`. Because parameter names and semantics vary across models, we use an LLM in agent mode to generate modified config.json files: given the model's Hugging Face repository URL and the name of a target kernel, the LLM inspects the model code, references the example config.json, and produces a new config.json file. Not all generated config.json files are valid. For example, the model in Figure 2a requires the activation function `silu`, and changing the `hidden_act` field in config.json will cause an assertion error at runtime. We simply discard any modified config.json that results in a runtime failure.

Second, framework environment variables and startup parameters also affect which kernels are used. We maintain a one-time, framework-specific mapping from each CUDA kernel to its required framework configuration. To trigger a kernel, we retrieve and apply its configuration from this mapping. If a kernel requires coordinated changes to both framework settings and config.json, we apply the framework settings first, then ask the LLM to modify config.json.

4.4 Interface Constraint Generation

Given a configuration, HFProbe executes the model with multiple combinations of batch size and sequence length, whose product equals the number of input tokens. These two parameters are varied because they are controlled by users. HFProbe then uses the recorded dynamic information to construct the implicit interface between the model and each called CUDA kernel as the conjunction of a set of symbolic constraints. Because the same kernel may be invoked under different call contexts within a single forward pass, HFProbe uses call-stack information to group invocations and infers constraints independently for each group.

For each kernel, HFProbe builds interface constraints over its wrapper function's arguments, including tensor dimensions, tensor value ranges, and primitive parameters. These constraints fall into four categories.

Model-fixed constants. Some arguments are determined by the model architecture (*e.g.*, hidden dimension). HFProbe identifies them by checking whether a value is invariant across all runs and constrains it to the observed constant.

User-controlled linear arguments. Among the arguments that change across runs, HFProbe isolates those that scale linearly with batch size, sequence length, or token count by verifying that the ratio to the corresponding value is constant across runs. These arguments are directly controllable by end users, and cuKLEE treats them as the primary source of attacker-controlled inputs during verification.

Inter-argument equalities. Kernel semantics impose dependencies among arguments. For example, a matrix multiplication requires the inner dimensions of its operands to match. HFProbe detects arguments that always share the same value and links them with equality constraints.

Residual upper bounds. Remaining arguments may still depend on the model architecture indirectly. HFProbe bounds them using the maximum value observed across all runs.

For example, after profiling the model in Figure 2a with a configuration from [23], HFProbe generates constraints stating that `out` and `input` in Figure 2c share the same shape, that their second dimensions are model-fixed at 896 (the hidden size), that their first dimensions are linear in the token count, and that `scale` contains exactly one element.

5 cuKLEE: Symbolic Execution on CUDA

As shown in Figure 1, cuKLEE begins its analysis of a CUDA kernel (*e.g.*, Figure 2d) from the kernel's wrapper function (*e.g.*, Figure 2c), collecting information about how the kernel is invoked. It then performs symbolic execution on the kernel to detect potential memory bugs. cuKLEE allows users to specify constraints on the wrapper's inputs, so that the interface constraints generated by HFProbe can be incorporated, improving both accuracy and efficiency of the symbolic execution. Furthermore, cuKLEE performs additional validation of detected bugs to ensure they can be triggered

Table 2. Constraint variables defined for tensor `t`.

Tensor Fields	Description	Variables
`t.storage_.data_ptr_`	base address	B_t
`t.numel_`	# of elements	N_t
`t.sizes_and_strides_.size_`	# of dimensions	D_t
`t.sizes_and_strides_`	dimensions	$d_t^0, d_t^1, \ldots d_t^{D-1}$
`t.data_type_.itemsize_`	element size	S_t

under the target inference framework and GPU environment. Overall, cuKLEE addresses three key challenges.

First, prior symbolic execution techniques (*e.g.*, KLEE [10]) struggle to handle buffers whose sizes are determined at runtime. In contrast, most CUDA kernels in LLM inference systems operate on the raw memory of tensor objects (*e.g.*, `out` and `input` on line 2 of Figure 2d), whose shapes vary depending on model architectures or user-provided tokens. How can cuKLEE analyze tensors with dynamic shapes and memory operations on the tensors?

Second, wrapper functions often call tensor methods to set up parameters (*e.g.*, `input.size(-1)` on line 2 of Figure 2c) for kernel launches. These methods often involve complex memory management, and analyzing them in full would significantly increase execution time and reduce precision. How can we summarize these tensor methods to make cuKLEE both efficient and accurate?

Third, CUDA kernels run across many threads, and analyzing each thread separately would result in redundant work. How can we share symbolic execution results across threads to avoid unnecessary computations?

5.1 Memory Model of cuKLEE

cuKLEE treats tensor memory separately from other memory regions.

Tensor Memory. All input tensors are stored in GPU global memory and allocated by the inference systems independently of the kernels. Therefore, kernel programmers cannot assume any specific memory layout among tensors. In practice, they never use a pointer to an element of one tensor to refer to an element of another tensor after arithmetic computations. We leverage this property by modeling tensors' underlying memory regions as discrete and far away from each other.

This design contrasts with KLEE's memory model, which treats the entire heap as a single continuous memory space. In KLEE, if a buffer has a symbolic size, the base addresses of all subsequent buffers become symbolic as well, making memory-access reasoning intractable. This is the fundamental reason why KLEE cannot handle buffers whose sizes are determined at runtime.

cuKLEE's modeling strategy greatly simplifies the detection of out-of-bounds accesses on tensor memory. Because each tensor occupies a distinct memory region, every pointer derived from a tensor carries constraints that relate it to that tensor's base address. When a pointer is dereferenced, cuKLEE uses these constraints to identify the source tensor and check the access against its bounds directly, without scanning all allocated memory objects—as a conventional symbolic execution engine (*e.g.*, KLEE) must do to determine which object a pointer may reference. For example, when analyzing line 14 in Figure 2d, cuKLEE immediately determines that the memory write must be checked against the bounds of `out`. This follows the constraint established on line 9, which binds `v_out` to the base address of `out`.

Table 3. Tensor-method handling examples. *('-': ignored.)*

Category	Tensor Methods	Constraints
I	`t.numel()`	N_t
I	`t.data_ptr()`	B_t
II	`t2=t1.sum(0)`	$N_{t2} = N_{t1}/d_{t1}^0$, $D_{t2} = D_{t1} - 1, S_{t2} = S_{t1}$, $d_{t2}^0 = d_{t1}^1, \ldots, d_{t2}^{D-1} = d_{t1}^D$
III	`check_dim_size (t, n, i, exp)`	$D_t = n, d_t^i = exp$
IV	`t.toString()`	-

Other Memory. For global and local memory allocated inside wrapper and kernel functions, we use the contiguous memory model adopted by KLEE. CUDA shared memory supports both dynamic allocation at kernel launch sites and static allocation inside kernels, and cuKLEE handles these two cases separately. For dynamically allocated shared memory, cuKLEE infers a concrete or symbolic size from the kernel launch parameters. For statically allocated shared memory, cuKLEE obtains the fixed size directly from the kernel declaration. cuKLEE models global, shared, and local memory as separate regions and resolves pointer targets only within the region to which they belong.

5.2 Handling Tensor Methods

To handle tensor methods, cuKLEE associates each tensor with a set of symbolic constraint variables, representing its base address, element count, dimensions, and other properties in Table 2. Since the number of dimensions may remain unknown throughout the analysis, cuKLEE maintains a map to record all known dimensions, where the keys are dimension indices and the values are the corresponding symbolic variables (*e.g.*, d_1, d_2). For example, `input.size(-1)` on line 2 of Figure 2c accesses the last dimension of tensor `input`. cuKLEE processes this method by adding an entry to the map with key -1 and value d_{input}^{D-1}.

With the constraint variables, cuKLEE supports 140 `Tensor` and `TensorIterator` methods, grouped into four categories. Table 3 shows examples. ① *Property queries* (36 methods) return tensor metadata such as element count or shape. cuKLEE models their return values using the symbolic variables in Table 2. For example, `input.numel()` on line 3 of Figure 2c returns N_{input}. ② *Tensor-producing methods* (70 methods) create a new tensor or a sub-tensor. cuKLEE introduces a set of fresh symbolic variables for the result and

Table 4. Constraint variables defined for CUDA threads.

Variables	Description	Type
blockIdx.x/y/z	block identifier	ID
threadIdx.x/y/z	thread identifier	ID
gridDim.x/y/z	grid size	launch
blockDim.x/y/z	block size	launch

adds constraints relating it to the original. For example, `t2 = t1.sum(0)` reduces `t1` along its first dimension, so `t2` has one fewer dimension and its i-th dimension equals the $(i{+}1)$-th dimension of `t1` (Table 3, Row 3). ③ *Property checks* (4 methods) inspect tensor properties (*e.g.*, contiguity); cuKLEE encodes the success condition as a constraint. ④ *Analysis-irrelevant methods* (30 methods) do not affect bug detection and are skipped.

5.3 Handling CUDA Threads

CUDA kernels execute across thousands of threads organized into blocks and grids. All threads run the same code but diverge based on their thread and block identifiers, which programmers also use to partition work over input data.

Rather than enumerating threads, cuKLEE models thread and block identifiers as symbolic variables, so that a single symbolic execution pass captures the behavior of all threads. As shown in Table 4, cuKLEE introduces six ID variables (*e.g.*, `threadIdx.x`, `blockIdx.y`) and six launch variables (*e.g.*, `blockDim.x`, `gridDim.y`). ID variables are bounded by launch variables following CUDA semantics (*e.g.*, $blockIdx.x < gridDim.x$), and launch variables are concretized or constrained when cuKLEE analyzes kernel launch sites (*e.g.*, $gridDim.x = num\_tokens$ in Figure 2c). With this approach, thread-independent bugs are detected as in sequential programs; thread-dependent bugs (*e.g.*, line 13 in Figure 2d) are captured through constraints over the ID variables.

For loops that distribute work across threads (*e.g.*, the highlighted loop in Figure 2d), cuKLEE analyzes their loop bodies once and then adds the loops' exit conditions to its execution states (*e.g.*, $i \geq num\_vec\_elems$) for the remaining execution. This suffices because such loops follow a simple strided pattern: each thread starts at an offset given by its global thread index and advances by the thread count. A single symbolic analysis of the loop body can capture every loop index a thread may process, while the exit condition characterizes the post-loop state.

To detect data races, cuKLEE introduces a second set of six ID variables representing a distinct thread. It asserts that the two threads differ by negating the conjunction of pairwise equality over their ID variables. For each global-memory access, cuKLEE checks whether the second thread may perform a conflicting write to the same address. cuKLEE performs a similar check for shared memory but additionally constrains the two threads' block ID variables to be equal, since shared memory is private to each block.

5.4 Synthesizing Input and Buggy Constraints

Input Constraints. Before analyzing each wrapper function, cuKLEE introduces three symbolic variables: *BS* (batch size), *SL* (sequence length), and *TC* (token count), related by $TC = BS \times SL$. In addition to the interface constraints from HFProbe, cuKLEE bounds these variables using two extra sources: ① *Model-defined limits.* Many models specify a maximum sequence length through fields such as "`max_position_embeddings`", "`n_positions`", or "`max_seq_len`" in their config.json files. HFProbe extracts these values from each input model's config.json and cuKLEE uses them to upper-bound *SL*. ② *Practical input limits.* Even when a model does not specify a maximum, the sequence length is bounded in practice. Current production models (*e.g.*, GPT-5.4 [57], Claude Opus 4.6 [13], Gemini 3.1 [26]) support up to roughly one million tokens. We therefore impose default bounds of $BS \leq 1{,}000$ and $SL \leq 1{,}000{,}000$, which can be adjusted for specific deployment scenarios.

Buggy Constraints. cuKLEE checks four types of memory bugs (data-race detection is described in Section 5.3). For *out-of-bounds accesses*, cuKLEE adds a constraint at each memory access via a pointer asserting that the pointer falls outside the valid range of the accessed buffer. For *integer overflows*, cuKLEE adds a constraint at each 32-bit addition or multiplication asserting that the result exceeds the type's maximum value. For *null-pointer dereferences*, cuKLEE adds constraints for each pointer dereference indicating the pointer may have been computed from an unknown base pointer. All input tensors are assumed to be valid with base addresses properly initialized by the inference frameworks.

5.5 Bug Detection and Validation

cuKLEE begins symbolic execution from each wrapper function under the input and interface constraints, processing tensor methods (Section 5.2) and, upon each kernel launch, introducing thread ID variables and launch variables constrained by CUDA semantics (Section 5.3). cuKLEE then executes the kernel body instruction by instruction. At each potentially buggy instruction, it synthesizes buggy constraints (Section 5.4), conjoins them with existing constraints, and invokes a solver. A satisfiable result yields concrete values for all symbolic variables, including batch size *BS* and sequence length *SL*.

However, not every satisfiable assignment is triggerable in practice. *BS* and *SL* are end-user inputs to the inference framework, but the framework may reject certain values, or the resulting memory footprint may exceed GPU capacity. cuKLEE checks both conditions to confirm a bug.

We replay the (BS, SL) pair via HFProbe's profiling backend. If the kernel is still invoked and its arguments match the interface constraints, the bug is confirmed to be reachable through the framework.

Table 5. Benchmark information and detection results. *(IO: integer overflow, OOB: out-of-bounds, DR: data race, and NULL: null-pointer dereference. x_y: x true positives, y false positives. '-' indicates zero true and false positives.)*

	Models	Kernels	IO	OOB	DR	NULL	Total
vLLM	62	50	7_2	19_5	0_2	-	26_9
HF	54	117	131_0	-	-	-	131_0
Papers	4	6	24_0	-	-	-	24_0
Total	120	173	162_2	19_5	0_2	-	181_9

We estimate GPU memory consumption as model-weight storage plus KV-cache size, scaled by a 1.25× fragmentation factor for the vLLM backend and 1.45× for Transformers. Model-weight size is obtained directly from the weight files. KV-cache size is $2 \times L \times TC \times H \times sizeof(dtype)$, where L is the number of layers, TC is the token count, H is the hidden dimension, and *dtype* is the cache data type. Intermediate tensors and temporary buffers are negligible relative to weights and KV-cache and are omitted. If the estimated total exceeds the target GPU's memory capacity, the bug is marked as untriggerable. For the bug in Figure 2d, the solver reports BS=116 and SL=18,079; the estimated memory usage is 211.2GB, well within a B300 GPU's 288GB memory capacity. We confirm the bug is triggerable on this hardware.

6 Evaluation

Implementation. HFProbe is implemented in Python. It performs call-graph analysis using the built-in AST module [65], provides two profiling backends (vLLM-0.9.0 [35] and Transformers v4.52.1 [25]), and employs ChatGPT agent [56] to mutate config.json files. The agent has web search capability to retrieve relevant code. In total, HFProbe comprises 18,453 lines of code, including both standalone modules and framework modifications. cuKLEE is built on KLEE v3.1 and operates on LLVM IR generated from CUDA source code. We implement 19,572 lines of C++ to support tensor abstractions and enable symbolic execution on CUDA threads and GPU instructions. cuKLEE uses Z3 [18] as its constraint solver.

Research Questions. Our experiments aim to evaluate M2K's effectiveness in detecting memory bugs in CUDA kernels used by real-world inference systems (Section 6.1), its coverage of CUDA memory bugs (Section 6.2), its advantages over existing techniques (Section 6.2), and the rationale behind combining dynamic model profiling with CUDA-specific symbolic execution (Section 6.3).

Experimental Settings. Most experiments run on a server with an Intel Xeon Silver 4110 CPU, 256GB RAM, and Red Hat Enterprise Linux 9. We compile CUDA code for CUDA 12.1 using clang++ 13.0.0 with "`-emit-llvm`" to generate LLVM IR for cuKLEE. Unless stated otherwise, we target an NVIDIA B300 GPU with 288GB memory for bug validation.

6.1 Bug Detection in the Wild

6.1.1 Methodology. As shown in Table 5, we evaluate M2K on models and kernels from three real-world sources. First, we extract all text-generation models and their associated kernels from vLLM-0.9.0, the latest release at the start of this project, yielding 62 models and 50 kernels, with substantial kernel reuse across models. Because vLLM omits *config.json* files, we retrieve them from Hugging Face by querying model IDs specified in vLLM's tests, and selecting the config.json from the top-ranked repository for each model. Second, we gather all Hugging Face models that contain custom CUDA kernels, resulting in 54 models and 117 unique kernels. Third, we survey top-tier AI conference papers from the past two years and identify four [32, 50, 61, 76] whose artifacts provide both Hugging Face models and custom kernels, contributing four models and six kernels. In total, our evaluation includes 120 models and 173 kernels, comprising 170,875 lines of Python model code and 139,951 lines of CUDA code.

We evaluate benchmarks from the first source using the vLLM backend and the remaining benchmarks using the Transformers backend. For each model, we attempt configuration mutation only once for any CUDA kernel that is statically reachable but not triggered by the collected config.json. For every configuration (both original and mutated), we test nine combinations of batch sizes $\{1, 3, 5\}$ and sequence lengths $\{1, 7, 17\}$, chosen to keep profiling short while remaining realistic and uncommon. Each CUDA kernel is analyzed by cuKLEE for up to one hour, following common practice in prior symbolic-execution work [10, 34, 47].

For each bug reported by cuKLEE, we examine the corresponding CUDA code to verify that the reported tensor shapes and input values indeed trigger the issue and are semantically valid. To ensure objectivity, each bug is independently reviewed by at least two authors. Because vLLM kernels are shared across models, we count unique bugs by kernel line number, even if they are triggered by different models. For out-of-bounds accesses resulting from a preceding integer overflow, we count only the integer overflow.

6.1.2 Experimental Results. As shown in Table 5, M2K detects 181 memory bugs in CUDA kernels, including 162 integer overflows and 19 out-of-bounds accesses. It also identifies one benign race in vLLM, which we conservatively classify as a false positive. M2K does not report any null-pointer dereferences. The large number of bugs demonstrates *M2K's effectiveness in detecting memory bugs in CUDA kernels.*

In addition, M2K detects bugs across benchmarks from all three sources, underscoring *its ability to analyze diverse models and kernels.*

We have reported all 26 bugs detected from vLLM, which has an actively maintained GitHub repository. Of them, eight have been fixed in the most recent version, and four extra bugs have been confirmed by the developers. Among

```
1  // Qwen-1_8B-Chat: heads=16, height=128, hidden_size=heads*height
2  // batch: batch_size, width: seq_len
3  __global__ void VecQuant8BatchMatMulKernel_faster_old(const
   half* vec, const uint8_t* mat, half* mul, int batch, int width,
   int heads, int height) {
4    int weight_total = batch * width * heads * height; // overflow
5    int h = 128 * blockIdx.x;
6    int w = 128 * blockIdx.y + threadIdx.x;
7    int i = width * h + w;
8    half2 weight[64];
9    for (int b = 0; b < batch; ++b) {
10     for (int head = 0; head < heads; ++head) {
11       int shift = b * heads + head;
12       for (int k = 0; k < 64; ++k) {
13         int index1 = shift * height * width + i + (2 * k * width);
14         if (index1 >= weight_total || w >= width || ...)
15           weight[k] = __float2half2_rn(0);
16         else
17           weight[k] = compute_weight(mat, ...);
18   }}}
19   compute_mul(mul, vec, weight,...);
20 }
```

Figure 4. A detected integer overflow bug.

the 18 unresolved bugs, the buggy code has been refactored for 13, while the other five are still in the codebase. We do not report bugs in Hugging Face models or research papers when the corresponding GitHub repository is unavailable, has been inactive for more than a year, or no contact email is found. For the 36 remaining cases, we have contacted the authors by email but haven't received any response.

Integer Overflows. Three detected integer overflows occur on unsigned 32-bit integers, and the rest occur on signed `int32` values. In 15 cases, the overflowed value is later used as an index, triggering subsequent out-of-bounds accesses. Figure 2d shows one such example.

In terms of what is being computed, 36 of the overflows come from multiplying the token count by the hidden size. For example, the kernel `VecQuant8BatchMatMulKernel_faster_old()` in Figure 4 multiplies input matrices `vec` and `mat`, and stores the result in `mul`. Since `mat` is quantized, the kernel dequantizes it on lines 9–18 before multiplying on line 19. During dequantization, `weight_total` is used on line 14 to determine whether a value should be zeroed. However, the computation of `weight_total` can overflow. Specifically, `batch * width` equals the token count, and `heads * height` equals the hidden size. While the hidden size is fixed at $16 * 128$, overflow can occur when the batch size is 128 and the sequence length is 8,192. When this happens, `weight_total` becomes negative, zeroing all dequantized values on line 15 and corrupting the subsequent computation. Another 98 integer overflows occur when computing the element count in $QK^\top$, whose size is `batch_size` × `num_heads` × `seq_len`^{2} and can exceed the `int32` limit. Four additional overflows arise when multiplying the token count by 1,024 (thread count), as shown in Figure 2d. The remaining 24 overflows come from a single code fragment that appears 24 times at different stages of a long computation. In each fragment, a 32-bit integer is shifted left by 10 bits and then cast to a 16-bit integer. When the original value exceeds 31, an overflow occurs during the cast. Neither the

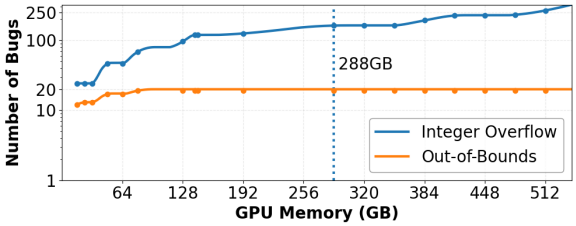

Figure 5. How the number of bugs changes with GPU memory. *(The vertical line represents NVIDIA B300.)*

model code nor the CUDA kernel imposes any constraint to prevent this value from exceeding that limit. We count these bugs as separate bugs, according to the methodology described in Section 6.1.1.

The proportion of integer overflows among the detected bugs is markedly higher than in the bugs collected from vLLM's change log in Section 2.3. We hypothesize that integer overflows are often under-reported because overflowed values are typically used in computations rather than as indices, causing silent result corruption instead of crashes.

Buffer Overflows. Eight buffer overflows occur when data is partitioned into blocks, but the final block is only partially filled, yet kernels access it as if fully populated. The remaining 11 arise when input tensor values are used to compute indices into another tensor without adequate bounds checking. For them, we further confirm that the framework code permits the buggy tensor content to pass into the kernel.

Data Races. M2K detects a data race, where an array is used to count tokens assigned to each expert, with one array element per expert. The array elements are partitioned among warps. During initialization, all 32 threads in a warp write zero to all the elements in the warp's partition, creating a race. We consider the race benign and classify it as a false positive, as all conflicting threads write the same value to the same location. Nevertheless, this case demonstrates M2K's capability to detect data races in CUDA kernels.

Security Impacts. As an initial exploration, we reproduce a subset of the detected bugs and examine their potential security impacts. A systematic study is left for future work. Most reproduced integer overflows corrupt computation results and may affect downstream tasks. Most reproduced out-of-bounds accesses (*e.g.*, Figure 2d) trigger crashes because they access unmapped memory. We also find one case in which a signed-index overflow is used in a memory write, modifying a neighboring tensor and corrupting the inference results of other users in the same batch. We record a video demonstrating this effect [29]. In a newer version of vLLM, we find a similar bug in which an overflowed unsigned index is used for a memory read, exposing another user's inference output. This bug was introduced after vLLM-0.9.0 and is therefore not included in Table 5. Our finding leads to a published security advisory [73], and we also record a demonstration video [28]. Finally, we find it difficult to exploit the detected out-of-bounds accesses for arbitrary code

Table 6. Comparison with baseline techniques. *(20 bugs are for each row. '/': not applicable.)*

	Honeycomb	GKLEE	ESBMC-GPU	cuKLEE
vLLM	/	/	/	15
Simplified	0	5	3	19

execution, since return addresses are typically stored in registers during CUDA function calls and are not easily overwritten unless register spilling occurs.

False Positives. M2K reports nine false positives (false discovery rate of 4.74%), arising from three sources. First, six stem from HFProbe's inability to track tensor values across kernels: it returns zero-value tensors for kernel binding functions, so subsequent kernel bindings cannot track valid value ranges for those tensors, causing two integer-overflows, three out-of-bounds, and one data-race false positives. Second, two involve kernels that index tensors using values loaded from model weights. As HFProbe does not load the actual weights, it cannot accurately capture the indices; manual inspection confirms that the true weight values cannot trigger errors. Third, M2K identifies one benign race, as discussed earlier.

Memory Bugs vs. GPU Memory. Figure 5 shows how detected bugs vary with GPU memory size, which cuKLEE uses to decide whether a bug can be triggered. Out-of-bounds bugs plateau beyond 80GB, whereas integer overflows grow with larger memory. On the NVIDIA H100 (80GB), M2K finds 68 integer overflows and 19 out-of-bounds bugs; on the H200 (141GB), integer overflows rise to 119 while out-of-bounds bugs remain unchanged; on the B300 (288GB), integer overflows further increase to 162. This trend suggests that future GPUs with larger memory will expose more integer overflow bugs.

Time and Monetary Cost. M2K takes 1,073.62 total hours to complete its analysis: 1,032.99 for vLLM, 28.18 for Hugging Face, and 12.45 for research papers. Experiments run with multiple parallel processes, and the reported time reflects their aggregate. Using GPT costs $442.16. Considering that both the Python and CUDA codebases exceed 100,000 lines, M2K incurs moderate time and monetary costs.

More specifically, cuKLEE consumes 979.86 hours, substantially more than HFProbe (93.76 hours). HFProbe generates 1,274 distinct model configurations. Because many vLLM kernels are shared across models, we merge identical analysis contexts, yielding 8,562 unique contexts for cuKLEE. Kernel analysis completes within one hour for 7,784 cases and reaches the limit for 778 cases. All kernels exceeding the one-hour limit contain 600+ lines of CUDA code.

Answer to effectiveness: *M2K can accurately detect integer overflows and buffer overflows in CUDA kernels used within LLM inference systems.*

6.2 Coverage and Advancement

6.2.1 Methodology. We evaluate coverage using the 20 known vLLM bugs from Section 2.3. Since most of the bugs were reported by users without model information, we evaluate only cuKLEE. We retain inputs required by the model architecture, if any, and treat the rest as symbolic when running cuKLEE.

```
1  // num_thread = std::max(num_experts, 32)
2  // share_mem size: num_experts+1+(num_thread+1)*num_experts
3  __global__ void moe_align_block_size_kernel(int32_t num_experts){
4    extern __shared__ int32_t shared_mem[];
5  - int32_t* tokens_cnts = shared_mem + blockDim.x + 1;
6  + int32_t* tokens_cnts = shared_mem + num_experts + 1;
7    for (int i = 0; i < num_experts; ++i) {
8      tokens_cnts[num_experts * (threadIdx.x + 1) + i] = 0; // OOB
9    }}
```

Figure 6. An out-of-bounds bug from vLLM.

We compare cuKLEE with three static CUDA analysis tools: Honeycomb's validator [49], GKLEE [41], and ESBMC-GPU [62]. These tools cannot run directly on vLLM kernels due to incompatibilities with modern C++, CUDA, and PyTorch/vLLM libraries. Moreover, GKLEE and ESBMC-GPU require a `main()` function as their entry point, which vLLM kernels lack. To enable comparison, we simplify the vLLM kernels by rewriting modern CUDA and C++ features with legacy constructs, removing PyTorch and vLLM dependencies, and adding a `main()` function for each kernel that replaces the original wrapper and allocates GPU memory for tensors. The simplified kernels preserve the root causes of the bugs. Because Honeycomb targets HIP kernels [4], we additionally translate CUDA APIs to HIP equivalents using hipify-perl [3]. We do not compare with other related tools because they are not free [14, 59] or their code is not available [11, 40, 42, 45].

6.2.2 Experimental Results. As shown in Table 6, cuKLEE identifies 15 of the 20 known bugs, demonstrating *strong coverage of real memory errors in CUDA kernels*. Specifically, the detected bugs comprise nine integer overflows, five out-of-bounds accesses, and one null-pointer dereference, underscoring *cuKLEE's ability to detect the three bug classes.*

Figure 6 shows an out-of-bounds error. The wrapper function (omitted for brevity) allocates shared memory to record how threads assign tokens to experts. As shown on line 2, the first `num_experts + 1` entries store the total token count and the per-expert counts. The remaining entries form a 2D array storing per-thread, per-expert token routing plus one slot per expert for the expert's total. Line 5 attempts to skip this first region so that lines 7–9 operate on the second, but incorrectly uses the thread count (`blockDim.x`), instead of expert count, to compute the offset. As marked on line 1, when the expert count is below 32, the thread count exceeds it, leading to a buffer overflow on line 8. cuKLEE correctly reports this, with `threadIdx.x=31` and `num_experts=8`.

The missed bugs arise for four reasons. First, one null-pointer dereference occurs because the kernel accesses data on a different GPU, undetectable via static analysis. Second,

one out-of-bounds is caused by a mismatch between a tensor's declared shape and the underlying memory layout; cuKLEE assumes consistency between them and thus misses the issue. Third, two bugs exceed the one-hour analysis limit and remain undetected. Fourth, cuKLEE lacks support for C++ `std::map`, causing it to miss one bug that depends on this construct. For the simplified kernels, cuKLEE still misses the first bug but successfully identifies the other four.

Honeycomb detects no bugs because it only checks whether memory accesses stay within one of four coarse memory regions (protected, read-only, read-write, and private). All tested out-of-bounds accesses fall inside the read-write region, so Honeycomb cannot catch them.

GKLEE detects four out-of-bounds accesses and one integer overflow, but misses the rest for three reasons. First, seven bugs require `threadIdx.x` or `blockIdx.x` to exceed 30, but GKLEE assumes small constant values for both. Second, GKLEE fully unrolls loops, causing the analysis of seven bugs to hit the one-hour timeout. Third, GKLEE cannot detect the bug due to the kernel and its data on different GPUs.

ESBMC-GPU identifies two out-of-bounds accesses and one integer overflow. It misses nine, seven, and one bugs respectively for the same reasons as GKLEE.

Answer to coverage and advancement: *cuKLEE detects most of the tested known bugs, substantially more than the baselines, demonstrating its strong coverage of real CUDA memory bugs and its advantages over existing techniques.*

6.3 Rationality of Components

6.3.1 Methodology. We conduct an ablation study by disabling individual components of M2K and comparing each variant against the full system. We use the same vLLM models, kernels, and experimental setup as in Section 6.1, making the results directly comparable to those reported there. The variants are: ① *Without HFProbe.* We run cuKLEE directly on kernel wrapper functions without guidance from HFProbe. ② *Without cuKLEE.* We execute each model on real GPUs using both original and mutated configurations, and record runtime errors (*e.g.*, "illegal memory access"). Experiments are parallelized across four RTX 5090s and four H100s; since only error occurrence is checked, hardware differences do not affect results. ③ *Without configuration mutation.* We run M2K on the collected config.json files and frameworks' default configurations, without any mutations. ④ *Without validation.* All bugs reported by symbolic execution are treated as final outputs without validation.

6.3.2 Experimental Results. The full-featured version of M2K detects the most bugs in Figure 7, demonstrating *the value of each component.*

Removing HFProbe causes cuKLEE to report 2,349 false positives for two reasons. First, cuKLEE may infer tensor shapes that are theoretically possible but never occur in reality, leading to incorrect out-of-bounds reports. For example,

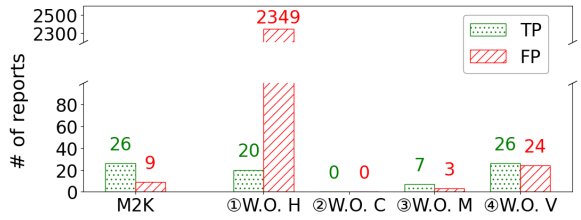

Figure 7. Contributions of M2K's components. *(26 bugs detected in vLLM. W.O.: without, C: cuKLEE, H: HFProbe, M: configuration mutation, and V: Validation.)*

in Figure 2c, cuKLEE incorrectly assigns tensor `scale` a zero size to trigger out-of-bounds access on line 4 of Figure 2d. However, this tensor always has one element in reality, making such a case infeasible. Second, cuKLEE may treat tensor shape values or other parameters as having extremely large, unrealistic values (*e.g.*, a hidden size of 2,147,483,640). Moreover, without model-architecture information, all tensor shapes become symbolic, forcing cuKLEE to explore far more possibilities. As a result, it fails to complete the analysis of 25 kernels within one hour, 18 more than when architecture information is available, and increases the average analysis time per kernel by 2.1×. The six missed bugs are due to this reason.

With cuKLEE disabled, no runtime errors are detected, so this variant reports nothing. Our experiments use a maximum batch size of 5 and a sequence length of 17 (only 85 tokens in total), far too few to expose hidden bugs. This shows that M2K uncovers issues that would otherwise appear only at large production scales.

Without configuration mutation, only 21 of the 50 CUDA kernels are exercised, leading M2K to miss 19 bugs.

Disabling validation causes M2K to report 15 more false positives. Five are cases where the token count is too large, causing memory usage to exceed the B300's memory size. The remaining ten cases should be filtered out by rerunning the profiling backend with the batch size and sequence length reported by the solver.

Answer to rationality: *Each component of M2K plays a distinct and essential role in its overall effectiveness.*

7 Limitations and Discussion

M2K currently targets Hugging Face models and CUDA kernels with tensor inputs, which together cover a large share of real-world usage. It profiles kernel usage without executing kernels on any specific GPU, making its effectiveness independent of any GPU hardware. M2K also supports GPU-dependent conditional compilation through cross-compilation and can handle fused kernels when the fused source code is available. At present, M2K does not support Triton kernels, NPU kernels, or TensorFlow models. However, applying its core idea of an explicit model-kernel interface to these settings is promising for uncovering potential issues.

M2K mutates execution configurations to trigger additional CUDA kernels. It does not aim to cover all possible models, nor is this necessary. In practice, users care about the specific model they deploy, rather than whether a kernel behaves correctly under every possible model. M2K therefore focuses on the kernel shapes exercised by a given model. Although our results show that the mutation is effective, it is far from a systematic inspection. Developing a more comprehensive inspection method remains future work.

In our experiments, cuKLEE analyzes each kernel for up to one hour. For certain kernels (*e.g.*, `gptq_marlin_gemm()`), not all paths are explored, potentially missing memory bugs. Since cuKLEE inherits KLEE's path exploration strategies designed for C/C++, optimizing them for CUDA code patterns is key to improving effectiveness.

Most bugs detected by M2K are integer overflows. Although such bugs look simple, we are not aware of a simpler way to resolve them. First, a static technique must track the numerical relations among variables, the path constraints under which these relations hold, and the constraints imposed by the model and the framework; without this information, it inevitably reports many false positives. Second, a dynamic technique must synthesize kernel inputs satisfying the model's constraints and run them on real GPU hardware, both of which are hard to automate. Moreover, as our experiments show, most overflows silently corrupt computation results, making them difficult to detect dynamically. Third, simply using 64-bit integers everywhere is not an ideal solution because it increases both runtime and memory overhead. M2K overcomes these difficulties by combining dynamic profiling with static symbolic execution, enabling it to detect integer overflows both effectively and precisely.

8 Related Work

Exploiting CUDA Memory Bugs. Prior work shows that CUDA memory bugs are highly exploitable [27, 66]. They can enable control-flow hijacking [19, 52], mind-control attacks [60], and side-channel attacks [53, 63, 80], leading to severe inference degradation or arbitrary code execution. These findings motivate our systematic analysis of CUDA kernels to detect memory bugs in them.

Static techniques have been developed to analyze CUDA kernels. ProofWright [11] verifies kernels against specifications generated by LLMs. COVT [46] translates CUDA into C/C++ and then detects integer and buffer overflows. Honeycomb's validator [49] ensures memory accesses remain within predefined coarse-grained regions. Tong *et al.* extend Dartagnan with GPU consistency models [72]. Methods based on symbolic execution have also been proposed [6, 40–42, 45, 62]. Overall, these static techniques struggle with dynamically sized memory buffers, do not support the widely used `Tensor` library, and cannot handle varied kernel launch configurations. We evaluate three of them [41, 49, 62] in Section 6.2 and find they are limited in usability and less effective than cuKLEE. Commercial static tools also exist [14, 59], but their technical details are unavailable for analysis.

Dynamic Techniques. NVIDIA provides several mechanisms to enforce GPU memory safety. The GPU MMU reports "illegal memory access" errors when unmapped virtual addresses are accessed [51]. CUDA-MEMCHECK [54] and its successor Compute Sanitizer [55] record metadata for GPU buffers, enabling detection of a broader range of memory errors at the cost of increased runtime overhead.

Researchers have also devoted efforts along this direction. GPUShield [37], GPUArmor [87], cuCatch [70], and LAK [77] build on principles similar to Compute Sanitizer, with the first three emphasizing lower runtime overhead and the latter using lock-key metadata to provide stronger memory safety. GMOD [20, 21] pads memory buffers and detects overflows by monitoring changes to the padding. IMT [69] and LMI [36] ensure GPU memory safety with ECC and unused upper bits of pointers, respectively. Race detection tools, including BARRACUDA [22], GRace [83], GMRace [84], and LDetector [44], track the last-accessing thread for each memory location and identify unsynchronized concurrent accesses. Despite these advances, these techniques rely on specific inputs to trigger bugs, require instrumentation or hardware support, and many incur large runtime overhead.

Fuzzing techniques have also been proposed for CUDA kernels and machine learning libraries [43, 48, 58, 68]. However, they overlook kernel usage context within LLM inference systems or target inference frameworks rather than CUDA kernels. They are ineffective in detecting memory bugs in CUDA kernels used in LLM inference systems.

9 Conclusion

This paper presents M2K, an automated tool combining dynamic model profiling with CUDA-specialized symbolic execution to detect CUDA memory bugs in LLM inference systems. M2K identifies 181 CUDA memory bugs in popular LLM inference systems when they perform inference with real models, demonstrating its practical effectiveness. Future work can improve the efficiency of symbolic execution for CUDA kernels and extend M2K to support NPU kernels.

Acknowledgments

We thank the anonymous reviewers and shepherd for their insightful feedback and valuable suggestions, and Yotta Labs for providing the compute resources. This work was supported in part by the U.S. National Science Foundation (NSF) under awards CNS-1955965 and CCF-2145394, and by the National Natural Science Foundation of China under grant 92582108. Linhai Song is the corresponding author.

References

[1] Eleni Adamopoulou and Lefteris Moussiades. 2020. Chatbots: History, technology, and applications. *Machine Learning with Applications* 2 (2020), 100006.

[2] Eleni Adamopoulou and Lefteris Moussiades. 2020. An overview of chatbot technology. In *Proceedings of the 16th IFIP International Conference on Artificial Intelligence Applications and Innovations (AIAI '20)*. Neos Marmaras, Greece.

[3] Advanced Micro Devices, Inc. 2025. hipify-perl. https://rocm.docs.amd.com/projects/HIPIFY/en/latest/hipify-perl.html.

[4] Advanced Micro Devices, Inc. 2025. What is HIP? https://rocm.docs.amd.com/projects/HIP/en/latest/index.html.

[5] Keqin Bao, Jizhi Zhang, Yang Zhang, Wenjie Wang, Fuli Feng, and Xiangnan He. 2023. Large language models for recommendation: Progresses and future directions. In *Proceedings of the 1st International ACM SIGIR Conference on Research and Development in Information Retrieval in the Asia Pacific Region (SIGIR-AP '23)*. Beijing, China.

[6] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: a verifier for GPU kernels. In *Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '12)*. Tucson, AZ, USA.

[7] Daniil A Boiko, Robert MacKnight, and Gabe Gomes. 2023. Emergent autonomous scientific research capabilities of large language models. *arXiv preprint arXiv:2304.05332* (2023).

[8] Rishi Bommasani. 2021. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258* (2021).

[9] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of artificial general intelligence: Early experiments with GPT-4. *arXiv preprint arXiv:2303.12712* (2023).

[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*. San Diego, CA, USA.

[11] Bodhisatwa Chatterjee, Drew Zagieboylo, Sana Damani, Siva Hari, and Christos Kozyrakis. 2025. ProofWright: Towards Agentic Formal Verification of CUDA. *arXiv preprint arXiv:2511.12294* (2025).

[12] Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[13] Claude. 2026. Models overview. https://platform.claude.com/docs/en/about-claude/models/overview.

[14] The Qt Company. 2025. CUDA Code Verification for Safety-Critical Systems. https://www.qt.io/quality-assurance/axivion-for-cuda.

[15] NVIDIA Corporation. 2025. CUDA C++ programming guide. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[16] Sumit Kumar Dam, Choong Seon Hong, Yu Qiao, and Chaoning Zhang. 2024. A Complete Survey on LLM-based AI Chatbots. *arXiv preprint arXiv:2406.16937* (2024).

[17] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems 35 (NeurIPS '22)*. New Orleans, LA, USA.

[18] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (ETAPS '08)*. Budapest, Hungary.

[19] Bang Di, Jianhua Sun, and Hao Chen. 2016. A study of overflow vulnerabilities on GPUs. In *Proceedings of the 13th IFIP WG 10.3 International Conference on Network and Parallel Computing (NPC '16)*. Xi'an, China.

[20] Bang Di, Jianhua Sun, Hao Chen, and Dong Li. 2020. Efficient buffer overflow detection on GPU. *IEEE Transactions on Parallel and Distributed Systems* 32, 5 (2020), 1161–1177.

[21] Bang Di, Jianhua Sun, Dong Li, Hao Chen, and Zhe Quan. 2018. GMOD: A Dynamic GPU Memory Overflow Detector. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT '18)*. Limassol, Cyprus.

[22] Ariel Eizenberg, Yuanfeng Peng, Toma Pigli, William Mansky, and Joseph Devietti. 2017. BARRACUDA: binary-level analysis of runtime RAces in CUDA programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*. Barcelona, Spain.

[23] Hugging Face. 2025. Qwen2-0.5B-Instruct/config.json. https://huggingface.co/Qwen/Qwen2-0.5B-Instruct/blob/main/config.json.

[24] Hugging Face. 2026. The AI community building the future. https://huggingface.co/.

[25] Hugging Face. 2026. Transformers. https://huggingface.co/docs/transformers/en/index.

[26] Google. 2026. Gemini 3.1 Pro Preview. https://ai.google.dev/gemini-api/docs/models/gemini-3.1-pro-preview.

[27] Yanan Guo, Zhenkai Zhang, and Jun Yang. 2024. GPU memory exploitation for fun and profit. In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security '24)*. Philadelphia, PA, USA.

[28] Mengting He and Linhai Song. 2026. TokenBleed: Cross-User Token Leakage via Integer Overflow in CUDA Kernels. https://youtu.be/F-SQ7ii7ob4.

[29] Mengting He and Linhai Song. 2026. TokenSmash: Exploiting CUDA Kernel Buffer Overflows to Manipulate LLM Inference. https://www.youtube.com/watch?v=ZhKBPK7BeZ0.

[30] David Holtz and Daniël de Kok. 2025. From Zero to GPU: A Guide to Building and Scaling Production-Ready CUDA Kernels. https://huggingface.co/blog/kernel-builder.

[31] Edmund Horner, David Fraser, Juha Jeronen, and Patrick Massot. 2020. pyan: Static call-dependency graph generator for Python. https://github.com/davidfraser/pyan.

[32] Wei Huang, Yue Liao, Jianhui Liu, Ruifei He, Haoru Tan, Shiming Zhang, Hongsheng Li, Si Liu, and Xiaojuan Qi. 2025. Mixture Compressor for Mixture-of-Experts LLMs Gains More. In *Proceedings of the 13th International Conference on Learning Representations (ICLR '25)*. Singapore.

[33] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515* (2024).

[34] Yunho Kim, Moonzoo Kim, Young Joo Kim, and Yoonkyu Jang. 2012. Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. Zurich, Switzerland.

[35] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th ACM SIGOPS Symposium on Operating Systems Principles (SOSP '23)*. Koblenz, Germany.

[36] Jaewon Lee, Euijun Chung, Saurabh Singh, Seonjin Na, Yonghae Kim, Jaekyu Lee, and Hyesoon Kim. 2025. Let-Me-In: (Still) Employing In-pointer Bounds Metadata for Fine-grained GPU Memory Safety. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1648–1661.

[37] Jaewon Lee, Yonghae Kim, Jiashen Cao, Euna Kim, Jaekyu Lee, and Hyesoon Kim. 2022. Securing GPU via region-based bounds checking. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22)*. New York, NY, USA.

[38] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems 33 (NeurIPS '20)*. Vancouver, BC, Canada.

[39] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. 2022. Solving quantitative reasoning problems with language models. In *Advances in Neural Information Processing Systems 35 (NeurIPS '22)*. New Orleans, LA, USA.

[40] Guodong Li, Ganesh Gopalakrishnan, Robert M Kirby, and Dan Quinlan. 2010. A symbolic verifier for CUDA programs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*. Bangalore, India.

[41] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P Rajan. 2012. GKLEE: concolic verification and test generation for GPUs. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. New Orleans, LA, USA.

[42] Guodong Li, Sreeranga P. Rajan, and Indradeep Ghosh. 2013. Symbolic execution and test generation for GPU programs. U.S. Patent US8595701B2. https://patents.google.com/patent/US8595701B2/en.

[43] Mingkai Li, Joseph Devietti, Suman Jana, and Tanvir Ahmed Khan. 2026. Challenges and Design Considerations for Finding CUDA Bugs Through GPU-Native Fuzzing. In *Proceedings of the 2nd Workshop on Ethical Systems and Architecture Design (HotEthics '26)*. Pittsburgh, PA, USA.

[44] Pengcheng Li, Chen Ding, Xiaoyu Hu, and Tolga Soyata. 2014. LDetector: A low overhead race detector for GPU programs. In *Proceedings of the 5th Workshop on Determinism and Correctness in Parallel Programming (WoDet '14)*. Salt Lake City, UT, USA.

[45] Peng Li, Guodong Li, and Ganesh Gopalakrishnan. 2014. Practical symbolic race checking of GPU programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. New Orleans, LA, USA.

[46] Wentao Li, Jianhua Sun, and Hao Chen. 2019. Detecting undefined behaviors in CUDA C. *IEEE Access* 7 (2019), 182559–182572.

[47] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering symbolic execution to less traveled paths. *ACM SIGPLAN Notices* 48, 10 (2013), 19–32.

[48] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23)*. Vancouver, Canada.

[49] Haohui Mai, Jiacheng Zhao, Hongren Zheng, Yiyang Zhao, Zibin Liu, Mingyu Gao, Cong Wang, Huimin Cui, Xiaobing Feng, and Christos Kozyrakis. 2023. Honeycomb: Secure and efficient GPU executions via static validation. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI '23)*. Boston, MA, USA.

[50] Vladimir Malinovskii, Denis Mazur, Ivan Ilin, Denis Kuznedelev, Konstantin Burlachenko, Kai Yi, Dan Alistarh, and Peter Richtárik. 2024. PV-Tuning: Beyond Straight-Through Estimation for Extreme LLM Compression. In *Advances in Neural Information Processing Systems 37 (NeurIPS '24)*. Vancouver, Canada.

[51] Microsoft. 2025. GpuMmu model. https://learn.microsoft.com/en-us/windows-hardware/drivers/display/gpummu-model.

[52] Andrea Miele. 2016. Buffer overflow vulnerabilities in CUDA: a preliminary analysis. *Journal of Computer Virology and Hacking Techniques* 12, 2 (2016), 113–120.

[53] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. 2018. Rendered Insecure: GPU Side Channel Attacks are Practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)* (Toronto, Canada).

[54] NVIDIA. 2022. CUDA-MEMCHECK. https://docs.nvidia.com/cuda/archive/11.4.4/cuda-memcheck/index.html.

[55] NVIDIA. 2025. NVIDIA Compute Sanitizer. https://developer.nvidia.com/compute-sanitizer.

[56] OpenAI. 2026. ChatGPT agent. https://chatgpt.com/features/agent/.

[57] OpenAI. 2026. GPT-5.4. https://developers.openai.com/api/docs/models/gpt-5.4.

[58] Shiwen Ou, Yuwei Li, Lu Yu, Chengkun Wei, Tingke Wen, Qiangpu Chen, Yu Chen, Haizhi Tang, and Zulie Pan. 2025. MirrorFuzz: Leveraging LLM and Shared Bugs for Deep Learning Framework APIs Fuzzing. *IEEE Transactions on Software Engineering* 52, 1 (2025), 360–375.

[59] Parasoft. 2025. NVIDIA CUDA and Parasoft. https://www.parasoft.com/integrations/nvidia-cuda/.

[60] 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), 102115.

[61] Yeonhong Park, Jake Hyun, SangLyul Cho, Bonggeun Sim, and Jae W. Lee. 2024. Any-Precision LLM: Low-Cost Deployment of Multiple, Different-Sized LLMs. In *Proceedings of the 41st International Conference on Machine Learning (ICML '24)*. Vienna, Austria.

[62] Phillipe Pereira, Higo Albuquerque, Hendrio Marques, Isabela Silva, Celso Carvalho, Lucas Cordeiro, Vanessa Santos, and Ricardo Ferreira. 2016. Verifying CUDA programs using SMT-based context-bounded model checking. In *Proceedings of the 31st ACM Symposium on Applied Computing (SAC '16)*. Pisa, Italy.

[63] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. 2016. CUDA Leaks: A Detailed Hack for CUDA and a (Partial) Fix. *ACM Transactions on Embedded Computing Systems* 15, 1 (2016), 1–25.

[64] pybind11. 2025. pybind11 (v3) — Seamless interoperability between C++ and Python. https://pybind11.readthedocs.io/en/stable/.

[65] Python. 2026. ast — Abstract syntax trees. https://docs.python.org/3/library/ast.html.

[66] Jonas Roels, Adriaan Jacobs, and Stijn Volckaert. 2025. CUDA, Woulda, Shoulda: Returning Exploits in a SASS-y World. In *Proceedings of the 18th European Workshop on Systems Security (EuroSec '25)*. Rotterdam, Netherlands.

[67] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. PyCG: Practical Call Graph Generation in Python. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*. Madrid, Spain.

[68] Saurabh Singh, Ruobing Han, Jaewon Lee, Seonjin Na, Yonghae Kim, Taesoo Kim, and Hyesoon Kim. 2026. CuFuzz: Hardening CUDA Programs through Transformation and Fuzzing. *arXiv preprint arXiv:2601.01048* (2026).

[69] Michael B Sullivan, Mohamed Tarek Ibn Ziad, Aamer Jaleel, and Stephen W Keckler. 2023. Implicit memory tagging: No-overhead memory safety using alias-free tagged ECC. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*. Orlando, FL, USA.

[70] 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. In *Proceedings of the 44th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '23)*. Orlando, Florida, USA.

[71] Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*

(2024).

[72] Haining Tong, Natalia Gavrilenko, Hernan Ponce de Leon, and Keijo Heljanko. 2024. Towards Unified Analysis of GPU Consistency. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*. San Diego, CA, USA.

[73] vLLM. 2026. Cross-User Data Leak Vulnerability. https://github.com/vllm-project/vllm/security/advisories/GHSA-7m6h-x95x-82q5.

[74] Hanchen Wang, Tianfan Fu, Yuanqi Du, Wenhao Gao, Kexin Huang, Ziming Liu, Payal Chandak, Shengchao Liu, Peter Van Katwyk, Andreea Deac, et al. 2023. Scientific discovery in the age of artificial intelligence. *Nature* 620, 7972 (2023), 47–60.

[75] Likang Wu, Zhi Zheng, Zhaopeng Qiu, Hao Wang, Hongchao Gu, Tingjia Shen, Chuan Qin, Chen Zhu, Hengshu Zhu, Qi Liu, Hui Xiong, and Enhong Chen. 2024. A survey on large language models for recommendation. *World Wide Web* 27, 5 (2024).

[76] Haoran You, Yipin Guo, Yichao Fu, Wei Zhou, Huihong Shi, Xiaofan Zhang, Souvik Kundu, Amir Yazdanbakhsh, and Yingyan Celine Lin. 2024. ShiftAddLLM: Accelerating Pretrained LLMs via Post-Training Multiplication-less Reparameterization. In *Advances in Neural Information Processing Systems 37 (NeurIPS '24)*. Vancouver, Canada.

[77] Chaochao Zhang and Rui Hou. 2022. LAK: A low-overhead lock-and-key based schema for GPU memory safety. In *Proceedings of the 2022 IEEE 40th International Conference on Computer Design (ICCD '22)*. Lake Tahoe, NV, USA.

[78] Gang Zhang and Jin Wuxia. 2019. Depends is a fast, comprehensive code dependency analysis tool. https://github.com/multilang-depends/depends.

[79] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223* (2023).

[80] Yu Zhao, Wenjie Xue, Weijie Chen, Weizhong Qiang, Deqing Zou, and Hai Jin. 2024. Owl: differential-based side-channel leakage detection for CUDA applications. In *Proceedings of the 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '24)*. Brisbane, Queensland, Australia.

[81] Zihuai Zhao, Wenqi Fan, Jiatong Li, Yunqing Liu, Xiaowei Mei, Yiqi Wang, Zhen Wen, Fei Wang, Xiangyu Zhao, Jiliang Tang, et al. 2024. Recommender systems in the era of large language models (LLMs). *IEEE Transactions on Knowledge and Data Engineering* 36, 11 (2024), 6889–6907.

[82] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. 2024. SGLang: efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems 37 (NeurIPS '24)*. Vancouver, BC, Canada.

[83] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. 2011. GRace: a low-overhead mechanism for detecting data races in GPU programs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*. San Antonio, TX, USA.

[84] Mai Zheng, Vignesh T. Ravi, Feng Qin, and Gagan Agrawal. 2014. GMRace: Detecting Data Races in GPU Programs via a Low-Overhead Scheme. *IEEE Transactions on Parallel and Distributed Systems* 25, 1 (2014), 104–115.

[85] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372* (2023).

[86] Yutao Zhu, Huaying Yuan, Shuting Wang, Jiongnan Liu, Wenhan Liu, Chenlong Deng, Haonan Chen, Zheng Liu, Zhicheng Dou, and Ji-Rong Wen. 2025. Large language models for information retrieval: A survey. *ACM Transactions on Information Systems* 44, 1 (2025), 1–54.

[87] Mohamed Tarek Ibn Ziad, Sana Damani, Mark Stephenson, Stephen W Keckler, and Aamer Jaleel. 2025. GPUArmor: A Hardware-Software Co-design for Efficient and Scalable Memory Safety on GPUs. *arXiv preprint arXiv:2502.17780* (2025).