StepStone: LLM-Based GPU Kernel Driver Fuzzing via User-Space Libraries

Xiaochen Zou*, Juefei Pu*, Arrdya Srivastav*, Jonathan Cox*, Zhengchuan Liang*,
Yuan Tan*, Xingyu Li*, Yilin Zhu* and Zhiyun Qian*
*University of California, Riverside
xzou017@ucr.edu, jpu007@ucr.edu, asriv033@ucr.edu, jcox033@ucr.edu, zlian064@ucr.edu,
ytan089@ucr.edu, xli399@ucr.edu, yzhu305@ucr.edu, zhiyunq@cs.ucr.edu

***Abstract*—GPU device driver fuzzing is an underexplored area. GPUs are directly accessible by untrusted users and represent a huge attack surface (e.g., NVIDIA driver has over a million lines of code). While continuous fuzzing has mitigated many kernel bugs within core subsystems, GPU device drivers have not received sufficient attention. SyzDescribe, a state-of-the-art tool for generating fuzzing interfaces for device drivers, significantly improves device driver fuzzing, but falls short given the complexity of GPU device drivers.**

In this paper, we observed that syscalls are not the only interface one can use to fuzz a device driver. In fact, user-space libraries provide an alternative interface, which can also be utilized for device driver fuzzing. Fuzzing user-space libraries has a number of advantages, such as more functionality-specific and user-friendly APIs, and comprehensive documentation providing valuable information about the API interfaces. To capitalize on this fact, we leverage Large Language Models (LLMs), which are highly effective in understanding and extracting information from both code and natural language (both exist in API documents). Therefore, we introduce StepStone, a novel approach that leverages LLMs to generate syzkaller descriptions for GPU libraries (e.g., CUDA, Vulkan), and fuzz these libraries to *indirectly fuzz* GPU kernel drivers. Our experiments show that StepStone achieves two to four times the level of coverage when compared with SyzDescribe, KernelGPT, and Moneta, when fuzzing NVIDIA, AMD, and Mali GPUs, respectively. Additionally, StepStone uncovered 11 new bugs in NVIDIA kernel drivers, demonstrating its effectiveness and efficiency in GPU driver fuzzing.

1. Introduction

Over the past several years, the Linux kernel has emerged as an important target of continuous fuzzing, leading to thousands of bugs, as documented by syzbot [38]. These achievements have primarily focused on common kernel components, e.g., networking and filesystems. However, vendor-specific device drivers represent an overlooked attack surface. Recent exploits [9], [10] have demonstrated that device drivers can be sources of security threats to the kernel.

The GPU drivers stand out as a crucial attack surface because: (1) They are by design accessible to normal users [7], [8], [18]; any vulnerabilities in the drivers can potentially lead to privilege escalation. (2) GPU drivers are ubiquitous: both desktops and servers need GPUs, e.g., for graphics or AI applications. (3) GPU drivers are becoming increasingly complex, driven by the AI boom and the rising complexity of each new hardware generation.

However, fuzzing device drivers presents unique challenges. All device drivers, regardless of their size and complexity, are exposed to user space through a common set of narrow syscall interfaces, e.g., `open` and `ioctl`. Given that GPU drivers can have over a million lines of code, it is challenging to identify and test the functionalities exposed through such narrow interfaces. Several recent works [34], [39], [50], [53] aim to fuzz device drivers by generating syscall descriptions [26] suitable for fuzzing. In particular, their approach focuses on deriving the descriptions by analyzing the source code of the device drivers. SyzDescribe [39] uses static analysis to identify driver entry points, extract data types, and `ioctl()` command values. KSG [50], on the other hand, uses symbolic tracing to collect precise data structures and branching conditions (to differentiate functionalities). KernelGPT [53] leverages the state-of-the-art Large Language Models (LLMs) to analyze kernel source code and automatically generate syscall descriptions. However, SyzDescribe struggles with accuracy due to the inherent imprecision of static analysis. KSG, while precise, lacks scalability to handle sophisticated GPU driver codebases. KernelGPT applies state-of-the-art LLM techniques, but it still falls short in recovering complex data dependencies due to the complexity of GPU drivers and accurate value ranges for parameters.

In this paper, we argue that the syscall interface is not the best interface for fuzzing GPU drivers. In fact, GPU manufacturers, such as NVIDIA, provide user-space libraries as an intermediary to interact with the GPU driver. Such libraries offer hundreds of well-documented functionality-specific APIs, which are much more convenient to understand and use for fuzzing.

To this end, we propose StepStone, a novel approach to GPU driver fuzzing by indirectly fuzzing its user-space libraries. To effectively extract and translate valuable library documentation knowledge into descriptions supported by syzkaller, we employ state-of-the-art Large Language Models (LLMs). Unlike KernelGPT [53], we avoid directly

applying LLMs to understand complex source code logic, which can span over a million lines of code. Instead, we propose to utilize LLMs to understand much more compact, developer-facing library API documentation. This is ideal for our task because the documentation contains extensive natural language and sample code snippets that demonstrate usage.

Our evaluation shows that StepStone achieves four times the coverage of SyzDescribe, Moneta, and KernelGPT on NVIDIA drivers, and reveals 11 previously unknown bugs in NVIDIA GPU drivers. We also show the generality of our solution in fuzzing AMD and Mali drivers. Our main contributions are summarized as follows:

- We introduce a novel approach to fuzzing GPU device drivers, leveraging library interfaces, which we argue to be a more suitable interface for fuzzing than the syscall interface.
- We developed StepStone, a GPU driver fuzzer that integrates an automated LLM-based description generation module for library APIs with a library-capable kernel fuzzer. We commit to open-sourcing StepStone for future research and development.
- Our evaluation demonstrates that StepStone works well with four different GPU libraries and three GPU drivers. It significantly outperforms the state-of-the-art device driver fuzzer, achieving two to four times code coverage and discovering 11 previously unknown bugs.

2. Background

2.1. GPU Software Stack

A Graphics Processing Unit (GPU) is a specialized processor originally designed to accelerate image rendering, video display, and parallel computing. To interact with the GPU hardware from kernel and user space, GPUs utilize both kernel-mode drivers and user-mode libraries.

GPU Kernel-Mode Driver. The kernel-mode driver serves as the bridge between the GPU hardware and the operating system. Major GPU manufacturers, such as NVIDIA, AMD, have developed their own kernel-mode drivers for Linux. On the other hand, mobile phone manufacturers such as Samsung, Google, often adopt Mali [17] GPU driver for their phone GPUs. These drivers are essential for all the GPU functionality, such as device initialization, power management, memory management, and command scheduling and execution. Additionally, GPU kernel-mode drivers serve as the interface between the kernel-level GPU operations and user-space programs, allowing users to interact with the GPU by sending system calls (e.g., `ioctl`).

GPU User-Mode Library. The user-mode library is a complement to complex kernel-mode drivers. These libraries provide a developer-friendly abstraction layer over raw system calls by segmenting GPU functionality into two primary domains: `graphics` and `computing`. For graphics: OpenGL [1] and Vulkan [22] are widely used user-space libraries. They are both open standards, meaning the GPU manufacturers can provide their implementations under a unified API format. Although OpenGL has historically been a popular choice, it is gradually being replaced by Vulkan due to Vulkan's superior performance and lower hardware overhead.

For computing: OpenCL [21] and CUDA [5] are the two leading frameworks. OpenCL, as an open standard, is platform-agnostic and supports a wide range of devices. CUDA, in contrast, is a proprietary library developed and maintained by NVIDIA, exclusively available for NVIDIA GPUs.

To accommodate the diversity of tasks within these domains, both `graphics` and `computing` libraries are further subdivided into specialized APIs. These APIs provide fine-grained control over GPU functionality. For instance, the CUDA library offers over 400 APIs, allowing developers to manage devices, execute CUDA kernel, allocate and deallocate memory.

2.2. Device Driver Fuzzing

Linux device drivers are accessed via an `open` system call, followed by syscalls like `ioctl` to allow user-space interaction.

Syzkaller Description. Fuzzing a device driver effectively requires addressing key challenges: providing precise entry definition (e.g., syscall), establishing accurate data dependencies (e.g., file descriptor between `open` and `ioctl`), interpreting the involved data structures, and determining the value ranges for mutable fields. Syzkaller, a state-of-the-art kernel fuzzer, addresses these four challenges by allowing users to define these aspects through a language called Syzlang. A complete syzkaller description consists of two primary files: the description body and its constant definitions.

A description body file (commonly ending with `.txt`) is divided into four main sections: ❶ **APIs/syscalls**, which specify names, arguments, and return values of functions. APIs serve as the fundamental units for seed generation. ❷ **Resources**, which define data dependencies across APIs/syscalls. For instance, a file descriptor from `open` can be defined as a resource and reused by `ioctl`. ❸ **Flags**, which define possible values for a given integer type. Flags are used extensively in APIs and syscalls. When combined with `switch` statements, flags enable programs to branch into multiple code paths, such as the `cmd` parameter in `ioctl`. ❹ **Structures**, which define the types used in the API/syscall section, primarily consisting of structure definitions and integer type casts. The values of structure fields or integers may be constant or mutable.

Figure 1 illustrates a simplified syzkaller description (the upper two figures i.e., *description.txt and description.txt.const*) and a generated seed (lower figure *example seed*): The `sock_tcp` resource, highlighted in red, establishes a dependency shared between `socket$inet_tcp` and `ioctl$sock`. In the generated seed, this is represented by `r0`. The `ifreq_ioctls` flag (defined on line 7) provides valid `cmd` values for `ioctl$sock`. The fuzzer

```
description.txt
# Resource
resource sock_tcp[int32]
resource ifindex[int32]

# Flag
ifreq_ioctls = SIOCGIFINDEX, SIOCSIFPFLAGS, ...

# Structure
type ifreq_t[ELEM] {
    ifr_ifrn    string["syz_tun", 16]  (in)
    elem                ELEM
}

# API
socket$inet_tcp(domain const[AF_INET], type const[
    SOCK_STREAM], proto const[0]) sock_tcp

ioctl$sock(fd sock_tcp, cmd flags[ifreq_ioctls], arg ptr[
    out, ifreq_t[ifindex]])
```

```
description.txt.const
SIOCGIFINDEX = 0x8933
SIOCSIFPFLAGS = 0x8934
...
```

```
example seed

r0 = socket$inet_tcp(0x2, 0x1, 0x0)
ioctl$sock(r0, 0x8933, &(0x7f0000000180)={'syz_tun',
    <r1=>0x0})
```

Figure 1: Syzkaller Description & Example Seed

randomly selects the value `SIOCGIFINDEX` (0x8933) for this field. The `ifreq_t` structure, highlighted in violet, includes the `elem` field. In the generated seed, the structure passes another resource, `r1` (representing `ifindex`), to receive the `elem` value defined on line 12, which is returned by the kernel.

Syscall Description Generation. The majority of built-in syscall descriptions in syzkaller are written manually [39]. Recent approaches instead statically or dynamically analyze source code or even kernel module binaries to automatically generate syscall descriptions [32], [33], [34], [39], [50]. DIFUZE [34] uses static analysis to identify the device filename, analyze `ioctl()` syscalls, retrieve valid `ioctl` commands, and recover related data structures. KSG [50] applies symbolic tracing to recover structures and values. SyzDescribe [39], the state-of-the-art, robustly identifies driver entry points, extracts syscall argument types and `cmd` values for `ioctl()`, and infers syscall dependencies. KernelGPT leverages LLMs to analyze data dependencies, type definitions, and value constraints in kernel source code, and generate syscall descriptions.

3. Motivation

GPU drivers are massive. AMD GPU drivers (`amdgpu`) have a codebase of approximately 250,000 lines [25] compared to NVIDIA's 1.25 million lines [19]. It is simply infeasible to manually create complete syscall descriptions. In one recent GPU fuzzing work [43], we found that the authors' manually written descriptions are largely incomplete, including only one extracted data structure, and 31 pseudo-

```
switch(cmd) {
case FITRIM:
    return gfs2_fitrim(filp, (void __user *)arg);
case FS_IOC_GETFSLABEL:
    return gfs2_getlabel(filp, (char __user *)arg);
}
```

```
if (is_driver_ioctl) {
    unsigned int index = nr - DRM_COMMAND_BASE;

    if (index >= dev->driver->num_ioctls)
        goto err_i1;
    index = array_index_nospec(index, dev->driver->
        num_ioctls);
    ioctl = &dev->driver->ioctls[index];
}

func = ioctl->func;
```

Figure 2: Traditional ioctl vs. drm_ioctl

syscalls for NVIDIA and AMD drivers in total. Automated syscall description generation is a promising direction [32], [33], [34], [39], [50], [53], which is in principle applicable to any kernel module, including GPU drivers. For example, SyzDescribe has extracted 64 data structures and generated 5,757 pseudo-syscalls through static analysis, substantially better than Moneta's manually written descriptions. However, it still faces several serious challenges, which motivate us to seek alternative solutions.

3.1. Challenges of Syscall Fuzzing

While there are a few solutions to automatically generate syscall descriptions and fuzz device drivers, we focus our discussion on two state-of-the-art solutions, i.e., SyzDescribe and KernelGPT, because of their superior performance.

Limited System Call Support. Due to the significant implementation efforts for modeling various system calls, SyzDescribe supports only `open()` and `ioctl()`, while KernelGPT supports 10 syscalls, such as `open()`, `bind()`, `poll()`, `ioctl()`, `sendmsg()`, `setsockopt()`, *etc.* These limited system calls suffice for many conventional device drivers. However, GPU drivers have more extensive interfaces, including additional syscalls such as `mmap`, `read`, `flush`, *etc.* These calls are essential for handling the complex functionality of GPU drivers. The inability to support these calls results in incomplete descriptions, leaving significant gaps in code coverage.

Insufficient `ioctl()` Modeling. SyzDescribe considers only common code patterns of `ioctl()`, which rely on `switch-case` or `if-else` statements to identify the `cmd` values; an example is shown in the top half of Figure 2 where two different driver functionalities are exercised corresponding to two different `cmd` values. However, certain GPU drivers, such as AMD's, employ a different code pattern. The bottom half of Figure 2 illustrates the difference. Instead of implementing its own `ioctl()` using `switch-case`, the device driver relies on an existing kernel function `drm_ioctl()`, which uses a lookup table

of function pointers. As will be shown later in the evaluation, this causes SyzDescribe to fail to recover any `cmd` values for the AMD GPU driver. Surprisingly, KernelGPT, an LLM-assisted description generation tool, also suffers from insufficient `ioctl()` modeling. For the AMD driver, the LLM struggled to extract `cmd` info from the provided `ioctl()` function source code because the lookup table was not within the context, and KernelGPT does not specifically retrieve such a table and provide it to the LLM. This results in incomplete descriptions with only a small portion of `ioctl()` covered.

Incomplete Data Recovery. Static analysis struggles to recover and solve value constraints that were collected along the path, which often resulted in imprecise value ranges for mutable fields. SyzDescribe inherits this limitation and chooses to represent values as generic integers without capturing constraints. For example, the parameter `proto` at line 16 in Figure 1 must be zero in the `sock$inet_tcp` call. SyzDescribe will ignore these constraints and present it as `int32`. This imprecision leads to significant inefficiencies in fuzzing, as test cases with invalid argument values will not reach deep code in the kernel. While brute-forcing valid values is feasible for smaller drivers, it becomes impractical for GPU drivers due to their massive codebases and the sheer volume of mutable fields. LLMs, on the other hand, lack sufficient guidance, such as training examples and documentations, which results in severe data loss. We noticed KernelGPT missed more than 60 data structures and type definitions in NVIDIA drivers, and the resulting description requires substantial efforts in repairing. We suspect the reason for incomplete data recovery is because NVIDIA driver is highly specialized and lacks enough documentation, unlike the general Linux kernel.

The difficulty of modeling kernel nuances causes these three major challenges, making SyzDescribe highly inefficient for generating meaningful descriptions of GPU drivers. Unfortunately, the difficulty of modeling kernel nuisances is not a unique problem with SyzDescribe, but a long-standing issue in the fields [40], [45], [55].

3.2. User-Space Library vs. Syscall Fuzzing

We observe that syscall fuzzing is not the only way to fuzz device drivers. One interesting alternative is to fuzz user-space libraries, which can indirectly fuzz the device drivers. We argue that fuzzing the libraries has the potential to overcome the limitations of current syscall fuzzing methods such as SyzDescribe and KernelGPT. We list a few advantages below:

Functionality-Specific Interfaces. GPU drivers provide a large number of functionalities multiplexed through a narrow set of syscall interfaces, e.g., `ioctl()`, `read()`, and `write()`. This is a major root cause of the difficulties of extracting useful information out of such complex syscall implementations, e.g., `cmd` values. In comparison, GPU manufacturers provide user-space libraries that abstract kernel driver details and turn them into functionality-specific interfaces or APIs that are much more user-friendly. For

> CUresult cuDevicePrimaryCtxGetState (CUdevice dev, unsigned int* flags, int* active)
> Get the state of the primary context.
>
> Parameters
> dev - Device to get primary context flags for
> flags - Pointer to store flags
> active - Pointer to store context state; 0 = inactive, 1 = active
>
> Description
> Returns in *flags the flags for the primary context of dev, and in *active whether it is active. See cuDevicePrimaryCtxSetFlags for flag values.

Figure 3: CUDA Documentation

example, the NVIDIA CUDA library provides over 400 APIs. These APIs will internally invoke syscalls such as `ioctl()` with the right arguments. In other words, it is easier to exercise the underlying functionalities in GPU drivers by fuzzing the library APIs.

User-Friendly Interface. Another advantage is that library APIs are designed for a broad range of users developing applications. As a result, these APIs are inherently well-designed and well-documented, unlike syscalls, which are not intended for direct user interaction. This means that it is no longer a requirement to perform complex and challenging static analysis or dynamic analysis to recover data types and value ranges, which cause the various issues in SyzDescribe. Furthermore, LLMs can leverage the knowledge compactly embedded in existing documentation without analyzing complex source code, which is often error-prone as we observed using KernelGPT. For example, Figure 3 explains the `cuDevicePrimaryCtxGetState()` API from CUDA documentation. It provides valuable information such as the value range for `active` is either 0 or 1, and the returned value is stored in pointer `*flags` (an out pointer). Such information already encodes the necessary knowledge for fuzzing, including API, dependency, data type definition, and value ranges.

4. Design Exploration

Fuzzing user-space libraries is a promising approach to indirectly fuzz kernel drivers. However, we also noticed several challenges of fuzzing GPU libraries.

4.1. Challenges of Fuzzing GPU Library

Limitation of Existing Library Fuzzer. There are several recent library fuzzers [15], [31], [41] that are shown to be effective at finding bugs in libraries. However, when attempting to apply them to GPU library fuzzing, we encounter several challenges. Most library fuzzers, such as libfuzzer [15] and FuzzGen [41], are incapable of fuzzing GPU user-space libraries due to their reliance on access to the source code of library implementations. These fuzzers require source code for various purposes, such as instrumentation [15]

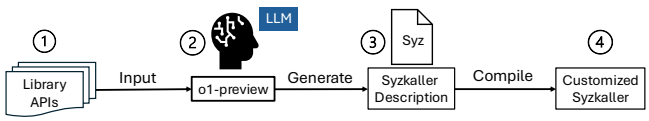


Figure 4: Rudimentary Design

or the construction of Dependence Graph [41]. However, most GPU user-space libraries, including CUDA, Vulkan, OpenCL, OpenGL, are closed-source, making it impossible to recompile them with the necessary modifications. It is worth noting that while Vulkan, OpenCL, and OpenGL are open standards, their actual implementations are developed by manufacturers and remain closed-source. This lack of access to source code prevents existing fuzzing tools from extracting useful fuzzing interface (API signatures, data structure, data dependency, and value range).

Although there exist library fuzzers, i.e., Hopper [31], which do not require source code and could theoretically be used for our purposes, we encountered several practical challenges in using it. First, Hopper and other library fuzzers run on the same machine where the library is. In contrast, fuzzing the kernel requires a separate environment (due to data loss after kernel crashing), e.g., syzkaller typically operates on a host machine and fuzzes the kernel in a guest. Therefore, we need to redesign Hopper to adopt a manager-executor architecture. Second, Hopper and other library fuzzers collect coverage feedback from the library as opposed to the kernel. While theoretically feasible, it is a significant undertaking to convert kernel coverage into the format suitable for library fuzzers to consume. Third, when experimenting with Hopper specifically, we find various limitations, e.g., not supporting `union` types [13], and a rough estimation shows that 30% of CUDA structures contain `union`, making hopper extremely difficult and inefficient to work with NVIDIA GPU libraries.

Last but not least, even if we can overcome the above challenges, it would be overly expensive to use Hopper for our purpose. This is because Hopper avoids any static analysis, which is well-known to be challenging. Instead, Hopper takes a dynamic approach of iteratively generating many test cases as trial-and-error in an attempt to learn what correct API descriptions should be. This is unfortunately overly inefficient considering library crash is contrary to the goal of exploring more kernel code.

Domain Knowledge Extraction. An efficient library fuzzer requires extensive domain knowledge of the library, including a complete list of API signatures, associated data structures, data dependencies and value constraints. Fortunately, GPU user-space libraries are typically accompanied by comprehensive documentation, providing all the information required to generate valid test cases. However, most of this knowledge is expressed in natural language, as shown in Figure 3, making it challenging to translate into a programmatic format.

4.2. LLM-based Library Interface Generation

To address the challenges of fuzzing GPU library, we proposed an LLM-based solution to automatically generate library interfaces to indirectly fuzz GPU drivers. Unlike KernelGPT, which works with a complex kernel source code interface and often makes mistakes, we opt for a more LLM-friendly interface, i.e., library APIs. As discussed in §2.2, a well-defined fuzzing interface should contain precise knowledge about API signatures, data dependencies, related data structure definitions, and value ranges for mutable fields. While library header files can provide details about API signatures and data structures, extracting accurate data dependencies and value ranges is far more difficult. For open-source libraries, their dependencies can be extracted via analysis of the dependency graph [50]. However, analyzing closed-source binaries like GPU libraries is significantly more challenging. Notably, they ship only header files as interfaces, but the implementation is closed-source.

Fortunately, advancements in Large Language Models (LLMs) offer a viable alternative. Equipped with exceptional natural language interpreting skills, LLMs supplied with library documentation can provide critical knowledge about data structures, dependencies, and value ranges, which are necessary for generating a good fuzzing interface. Moreover, the LLMs were trained by extensive internet data, which provides access to broader learning resources and code examples

4.3. Rudimentary Design and Insights

We propose a rudimentary design, illustrated in Figure 4. Unlike SyzDescribe, which extracts syscall descriptions, our approach aims to generate library interface descriptions with the aid of LLMs.

We randomly sampled 20 CUDA APIs and 20 Vulkan APIs, and grouped them based on their functionality (e.g., context management, device management), labeled as ❶. Then, we utilized a state-of-the-art LLM model, i.e., `o1-preview`, and supply it with the corresponding documentation (e.g., CUDA API Reference) to generate the descriptions (from ❷ to ❸).

To our surprise, the generated syzkaller descriptions were highly inaccurate. Numerous data types were incorrectly defined, incorrect syntax appeared in the generated descriptions, and several other issues. This result is simply unusable for fuzzing. After a thorough review, we categorized these errors into five distinct types.

1. Reasoning Degradation with Long Context Window. We observed significant reasoning degradation [37] when multiple APIs were provided to the LLM simultaneously. The reasoning degradation results in severe inaccuracies or even information loss in critical data structures, resources, or enumerations. In some cases, the LLM even omits certain APIs entirely. This issue may arise from the model's limited ability to process and retain context across multiple related inputs. When tasked with processing multiple APIs in the same context window, the LLM might struggle to inherit

```
1
2  CUresult CUDAAPI cuStreamWaitEvent(CUstream hStream,
       CUevent hEvent, unsigned int Flags);
3
4  typedef enum CUevent_wait_flags_enum}@> {
5      CU_EVENT_WAIT_DEFAULT  = 0x0,
6      CU_EVENT_WAIT_EXTERNAL = 0x1
7  } CUevent_wait_flags;
```

Figure 5: Implicit flags

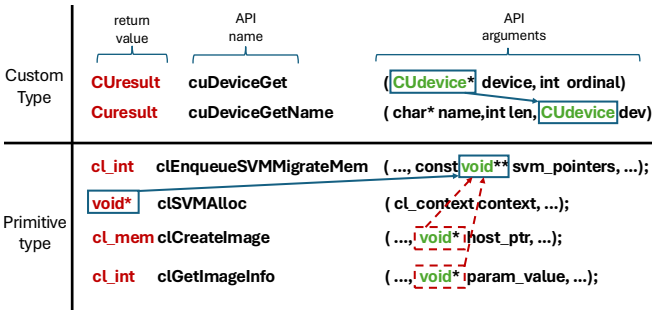


Figure 6: Data dependency

and maintain the knowledge, especially when processing complex APIs that involve many structures and enums. Another potential cause is the token limit within the LLM, which could result in truncation during processing.

2. Nested Structure Retrieval Failure. We noticed that the LLM often generates incomplete data structures in the description. Ideally, the LLM should recursively retrieve nested data types of arguments until it meets primitive types, such as `int`, `void`, *etc.* However, we found that the LLM struggles with such tasks and usually retrieves only the top layer of type or only a portion of nested type, causing an incomplete structure definition in generated description. This may be due to the complexity of the task, requiring the LLM to search for the relevant data type in the documentation and recursively retrieve all associated types. Additionally, type definitions in the documentation often mix code blocks with unstructured text and natural language, making it more challenging for the LLM to extract precise definitions.

3. Inaccurate Value Range for Primitive Type. We found many libraries uses primitive types in C to define API arguments or structures fields, rather than specifying explicit custom types, especially for `flags` arguments. This issue results in inaccurate value ranges for the arguments. For instance, as shown in Figure 5, the function `cuStreamWaitEvent` defines its third argument `Flags` as an unsigned integer. However, the API signature does not provide information about the possible values for this flag. In fact, we can only know the possible values for this `Flags` are defined in `CUevent_wait_flags` after reading the API documentation, but the LLM failed to extract this information. A more precise definition should use `CUevent_wait_flags` instead of `unsigned int`, as the former explicitly specifies two valid options: `CU_EVENT_WAIT_DEFAULT` and `CU_EVENT_WAIT_EXTERNAL`, whereas the latter introduces 2^{32} possibilities.

4. Dependency Misidentification. We noticed several data dependencies are missing in the LLM-generated description. LLMs effectively identify data dependencies when they involve the same custom types. For instance, as illustrated in Figure 6, the function `cuDeviceGet(CUdevice* device, int ordinal)` returns a `CUdevice` in its first argument, while the function `cuDeviceGetName(char* name, int len, CUdevice dev)` consumes `CUdevice` in its third argument. In such cases, the LLM correctly identifies a resource type for `CUdevice`, ensuring that the output from `cuDeviceGet`'s first argument is passed as input to `cuDeviceGetName`'s third argument.

However, identifying data dependencies becomes more challenging when they involve primitive-type or cross-type relationships. This is also a generic issue for prior works; Hopper only recognizes new dependencies from path coverage, which heavily relies on mutation quality. FuzzGen's dependency graph may recognize these kinds of dependencies, but it requires source code. Unfortunately, the LLM also struggles to recognize the dependencies due to a lack of precise dependency knowledge. Figure 6 shows another example. The function `clEnqueueSVMMigrateMem()` includes an argument `svm_pointers` of type `void**`, which is a generic pointer type that matches numerous other APIs that also use such types. However, the `svm_pointers` argument should specifically be derived from the return value of `clSVMAlloc()` according to the documentation [4]. As illustrated in Figure 6, this dependency should only happen with `clSVMAlloc()` and not other APIs such as `clCreateImage()` which also take such generic pointer types.

5. Syzlang Syntax Errors. In our experiment, we observed many syntax errors in the library API interface descriptions in the syzlang format. They require extensive manual efforts to fix, increasing the labor overhead and chances of introducing human errors. We suspect that it is because syzlang is a domain-specific language tailored for fuzzing. While very useful for our purposes, syzlang may not be well understood by the LLMs. Indeed, even though there are many existing syzkaller descriptions for the Linux kernel, the available syzlang syntax tutorial is incomplete and lacks comprehensive explanations (only a few pages) [26], compared to the widely adopted description languages like JSON and XML.

5. StepStone

The challenges identified in §4.3 reveal critical limitations of using LLMs generated library API descriptions. To address these challenges, we propose specific mitigation methods for each issue, as outlined in Figure 7. Unlike the rudimentary design, the improved workflow incorporates several new components to address the issues identified in our initial design.

5.1. Overview

As shown in Figure 7, we propose five novel components, labeled as ❶ to ❺, corresponding to the five

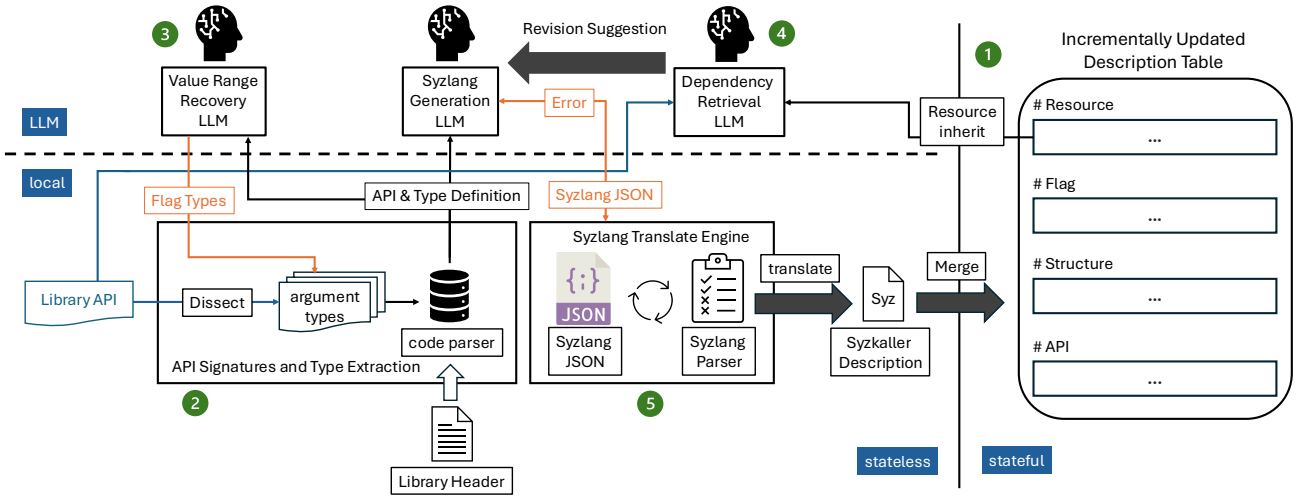


Figure 7: Workflow

challenges mentioned in §4.3, respectively.

At a high level, our solution works on a single API at a time. The generated description will be merged into the "Incrementally Updated Description Table" (component ❶). This table maintains accumulated descriptions over time, mitigating the risk of information loss between iterations. All other components remain stateless, ensuring that each API is treated consistently and without interference from others. Components ❷ to ❹ are used to extract the three key ingredients of API descriptions, i.e., API/type definitions, value range, and dependencies among APIs. These components interact with the "Syzlang Generation LLM" to produce the descriptions in an intermediate Syzlang JSON format. Finally, the JSON descriptions are then passed to the "Syzlang Translate Engine" (component ❺) and are translated into the final syzkaller descriptions. They are subsequently integrated into the "Incrementally Updated Description Table" (❶). The key insights of StepStone's new design are summarized as follows:

- **Selective stateful design**: StepStone's pipeline is made of predominantly stateless components (❷ to ❺) and a stateful description table (❶). This design enables StepStone to process hundreds of API inputs without exceeding token limits or being affected by other API results. The stateful description table ensures proper maintenance of the final output, avoiding duplication or omissions.
- **Modular design**: StepStone employs three specialized LLMs, each dedicated to specific tasks. This division of labor reduces the workload on the "Syzlang Generation LLM," allowing each LLM to focus on a narrower task scope while cooperating with one another. Specifically, two of the LLMs are solely responsible for extracting information and providing it to the "Syzlang Generation LLM".
- **Intermediate representation for syzkaller descriptions**: A customized JSON format is used to represent syzkaller descriptions, minimizing Syzlang syntax errors. This format is general and can also benefit other research [32], [33], [39], [50] that need to generate syzkaller descriptions.

5.2. Incrementally Updated Description Table

As illustrated in §4.3, LLM-generated syzkaller description suffers from severe information loss when sending in multiple APIs as input, due to token limitations and insufficient context maintenance capability of current LLM models.

To address this challenge, we design a hybrid state pipeline combining stateless components (❷ and ❺) and a stateful component(❶) to achieve a more reliable description generation, as shown in Figure 7. Specifically, we generate descriptions for a single API per request. The resulting syzkaller descriptions are then merged into a stateful "Incrementally Updated Description Table" (❶) that accumulates knowledge from each stateless iteration. During this merging process, the description table deduplicates entries to avoid redundancy, ensuring only new data are added to the table.

This design treats each API consistently without interference from others, eliminating the risk of discarding lower-priority data due to hallucination or truncation. However, this design may introduce problems with identifying cross-API dependencies, which will be discussed in §5.5.

5.3. API Signatures and Type Extraction

As discussed in §4.3, LLMs often struggle to recursively retrieve nested dependent data structures used by API arguments, leading to inaccurate descriptions. To mitigate this issue, we opt to directly supply the LLM with accurate type definitions and enforce strict adherence to the provided data. By doing so, we offload this error-prone task from LLM to a code parser, eliminating the risk of hallucination.

Our solution involves a lightweight code parser that operates only on the vendor-supplied C header files. These

```
API signatures
CUresult cuCtxCreate ( CUcontext* pctx, unsigned int
     flags, CUdevice dev )
```

```
LLM Response
{
    "integer_flags": [
        {"type": ["CUctx_flags"], "explain": "argument `
             unsigned int flags` in API `cuCtxCreate`
             should be `CUctx_flags` type according to the
              documentation. You should use `CUctx_flags`
             to replace `unsigned int`." }
    ]
}
```

Figure 8: Primitive type flag example

header files serve as interface specifications: they define the public APIs, types, and constants that must be shipped to developers so they can compile against the library, but they do not reveal the proprietary implementation logic, which remains in closed-source binaries.

When a new API is processed, we first extract its argument types and use the code parser to retrieve the corresponding type definitions. For data structures containing nested structures, our code parser supports recursive extraction of all nested structures. These extracted type definitions and the original API signatures are then sent to "Value Range Recovery LLM" ❸ for inspection of inaccurate value range. The final type definitions and API signatures will then be forwarded to the main "Syzlang Generation LLM".

5.4. Value Range Recovery LLM

As discussed in §4.3, the value ranges for primitive types can be missing, particularly for `flags` (e.g., `int flags`). Although the API and type definitions are clearly provided, they do not explicitly specify the value ranges for primitive-type flags, which are in fact `enum` types, as described in the documentation. To address this issue, we created a "Value Range Recovery LLM" using OpenAI assistant [20] with file search capability [12]. The file search capability allows us to search data within provided files without consuming tokens of the entire file, the details will be discussed in §6.1.

Specifically, we attached the entire API documentation (e.g., CUDA API references) to the file search of "Value Range Recovery LLM". The LLMs are then explicitly instructed to rely solely on this provided documentation rather than inferring or guessing from its internal knowledge base. We ask the LLM to search for primitive-type flags within the given input (an API or a collection of structure definitions), and return a JSON format that contains the identified flag types and their explanations. Figure 8 demonstrates an example of identifying a primitive-type flag from `cuCtxCreate()`. The "Value Range Recovery LLM" successfully recognizes the `unsigned int flags` is a primitive-type flag, and its value should come from an enum type `CUctx_flags`, based on the documentation's explanation. Figure 14 in Appendix presents the prompts we used for "Value Range Recovery LLM".

Revise your response based on the following statement (Your new response should only contain the revised json, no explanation is needed):
- Argument 'void* svm_ptr' from 'clEnqueueSVMMemFill' should be a resource because it comes from the return value of 'clSVMAlloc'. 'svm_ptr' is a pointer to a memory region that is allocated by 'clSVMAlloc'. The resource name should be 'clSVMAlloc_ret'

Figure 9: Dependency retrieval LLM revision suggestion

It is important to note that the "Value Range Recovery LLM" focuses exclusively on recovering value ranges for flag types. For other constraints (e.g., a parameter must be 0), we delegate decisions to the "Syzlang Generation LLM." Specifically, if the documentation defines a fixed value for a parameter, we instruct the LLM to adhere strictly to it. However, when the documentation specifies a value range for a non-flag argument, we currently choose not to impose the value range. This decision is based on our observations of kernel behavior. For instance, invalid flag values often lead to early returns (e.g., in `switch-case` handling), whereas invalid values for a size may trigger critical issues like out-of-bound access. Overly strict adherence to documented value ranges may result in degraded bug-finding ability.

5.5. Dependency Retrieval LLM

As discussed in §4.3, missing dependencies often happen in cases involving primitive-type and cross-type objects. While dependencies for custom types can be inferred directly from their type names (e.g., `CUdevice` in Figure 5), primitive or cross-type dependencies are typically only described in the API documentation, making them harder to detect. For example, the following sentence from OpenCL documentation describes a dependency between the input of `clEnqueueSVMMigrateMem()` (`svm_pointers` argument) and the output of `clSVMAlloc()`: "*svm_pointers is a pointer to an array of pointers. Each pointer in this array must be within an allocation produced by a call to clSVMAlloc.*".

To address this issue, we created an additional Assistant LLM specifically to extract such dependencies from the documentation. Similar to the previous LLM, we also provide this LLM with the complete API documentation (for file search). Specifically, we instruct the LLM to inspect the given API for relationships with other APIs: (1) whether its input depends on the output of another API (e.g., return value), and (2) whether its output will be used as input by any other API. We specifically asked the LLM to focus on cases where the types of input and output don't match. Figure 15 shows the instructions provided to the "Dependency Retrieval LLM."

If the LLM detects potential dependencies that are missing in the results, it sends a revision suggestion in natural language to the main "Syzlang Generation LLM", describing the names of new resources, the dependent APIs, and the

relationship between them (e.g., resource consumers and producers). Figure 9 demonstrates a revision suggestion from "Dependency Retrieval LLM" to "Syzlang Generation LLM". This revision suggestion will direct "Syzlang Generation LLM" to generate a new result that properly reflects this dependency.

If the dependency is a custom type such as `CUdevice`, we directly use the type name as the resource name because it won't have conflicts with other types. However, if a dependency is a primitive type (e.g., `void*`) or has a cross-type match, establishing a universal naming convention is challenging. A resource can originate from a return value, an argument, or even a structure field. A single API can return multiple resources. Therefore, we rely on LLM to determine the resource name based on its key properties: originated function name, structure type, return type (return value or arguments). For example, revision suggestion from Figure 9 indicates the LLM suggests to create a new resource called `ckSVMAlloc_ret` for `ckSVMAlloc`'s return value – the primitive type `void*`.

However, the stateless nature of the design can cause separate LLM sessions generate inconsistent names for the same resource (e.g., `ckSVMAlloc_ret_void`). To address this, each LLM session inherits all previously defined resource names by attaching them in the prompt, as shown in ❶. This ensures the stateless LLM remains aware of existing resources and avoids generating conflicting names for the same entity.

5.6. Syzlang Translate Engine

We observed numerous syntax errors in LLM-generated syzkaller descriptions. These issues are likely attributable to the poor documentation of Syzlang and the limited availability of relevant training datasets.

For example, the description in Figure 10 demonstrates one common error made by LLM. In Syzlang, if a `flag` is used as a structure field (`CUexecAffinityType` at line 4), its element type `int32` must be specified. However, if used as an API argument (line 8), the element type should be omitted. The LLM often incorrectly omits the type in a structure or includes it in an API argument, resulting in syntax errors.

One potential approach to address these errors is to prompt the LLM to fix each error iteratively based on error messages [44]. However, the Syzlang error messages are often unclear about the root causes, forcing the LLM to guess the underlying issues. For instance, the second argument of the `array` keyword is expected to be the length of the array (e.g., `array[int64, 4]`). An incorrect LLM-generated statement such as `array[int64, int32]` may produce the error message "*int32 is defined for none of the arches*". This message does not directly indicate the misuse of type `int32` as a length, potentially misleading the LLM into making unrelated or incorrect changes.

To overcome these challenges, we developed a "Syzlang Translate Engine". Instead of directly generating syzkaller descriptions, we instruct the "Syzlang Generation LLM" to

```
1  description
2
3  type CUexecAffinityParam[OP1] {
4          type    flags[CUexecAffinityType, int32]
5          param   OP1
6  }
7
8  syz_cuCtxGetExecAffinity(..., type flags[
       CUexecAffinityType])
```

```
1  JSON representation
2
3  {"type_name": "flags", "flags_enum": "
       CUexecAffinityType", "flags_element_type": "int32"}
```

Figure 10: Syzlang translation example

produce an intermediate JSON representation of the description. For example, the bottom half of Figure 10 shows the JSON representation of a `flag` type. The "Syzlang Generation LLM" was instructed to fill each field to complete the `flags` type regardless of whether it is inside a strucuture or an API. During the JSON representation translation, the "Syzlang Translate Engine" will decide whether to add the element type (`int32`) in the description based on whether it's in a structure or API arguments. To do so, "Syzlang Translate Engine" was implemented to have not only the capability of effectively decoding the customized JSON representation, but also the complete embedded knowledge of Syzlang syntax for translation. This design simplifies the task for the LLM, replacing the need to understand complex syntax nuisances (e.g., when to include the type), with a straightforward "fill-in-the-blanks" approach. In real-world cases, the JSON representations become complicated, the `flag` type might come from a structure `fields`, an API `params`, or even a pointer `ptr_object`. Figure 16 shows the complete JSON representations of structure `CUexecAffinityParam`. We also attached the complete predefined type schemas in Figure 17.

We noticed that the LLM occasionally makes mistakes in generating JSON format even though JSON is widely adopted. They are usually small issues like mistakenly inserting comments in the JSON. In practice, we find that these JSON errors have better messages indicating the error location, and can be easily corrected within a single iteration by prompting the LLM to revise its output.

6. Implementation

StepStone has two major components: LLM assisted description generation, and library-capable GPU fuzzer. We built LLM-based description generation with around 3,000 lines of Python, and added 2,800 lines in Go on top of syzkaller to build library-capable GPU fuzzer.

6.1. LLM-Based Description Generation

Syzlang Generation LLM & Auxiliary LLMs. The key requirement for choosing an LLM model is the file search

capability, which is fundamentally a form of retrieval augmented generation (RAG) [24]. Many latest LLM models at the time, such as Claude 3.5 Sonnet, Gemini 1.5 Pro, and Llama 3.1, only support uploading file to the prompt, which consumes tokens based on the size of the file. While, only OpenAI provides the "file search" as a built-in feature [12], which only consumes tokens when retrieving contents from the file. Since the main LLM, i.e., "Syzlang Generation LLM", requires strong reasoning and code generation skills due to the sophisticated tasks of distilling complex information and translating it into accurate Syzlang JSON. We found `o1-preview` significantly outperforms other OpenAI models. The two auxiliary LLMs, i.e., "Value Range Recovery LLM" and "Dependency Retrieval LLM" focus on retrieving specific knowledge from documentation and don't need strong reasoning and code generation skills. Therefore we choose the most cost efficient model, i.e., `GPT-4o`.

6.2. Library-Capable GPU Fuzzer

Library API Awareness. Generating test cases using library APIs is not natively supported by syzkaller. However, syzkaller inherently allows seed generation via system calls or using wrapper functions (a.k.a. pseudo-syscalls). A wrapper function starts with "syz_" will be treated as a new system call, and it is allowed to participate in seed generation and mutation. To enable library API fuzzing, StepStone automatically creates batches of wrapper functions, each corresponding to a library API. These wrapper functions act as intermediaries, passing arguments directly to the native library APIs, and send back the return value from the library APIs.

Fuzzing GPU drivers from VMs. Fuzzing the GPU drivers using virtual machines enables an easy way to monitor the VM status and capture kernel bugs. However, a discrete PCI hardware cannot simultaneously be accessible to both the host and the VM. We thus employ PCI passthrough to export the GPUs to VMs so that the fuzzer can access the GPU. Specifically, we enabled I/O virtualization and isolate the GPU into dedicated IOMMU group [2]. Next, we unbind the GPU's IOMMU group from the host machine and rebind it to the VFIO driver [28]. The VFIO driver exposes the isolated IOMMU group through a user-space interface located at `/dev/vfio/`, so the virutal machine can finally access the GPU hardware.

7. Evaluation

7.1. Experiment Setup

GPU Kernel Drivers & User-Space Libraries. We selected Nvidia, AMD, and Mali, three of the most popular GPU manufacturers across desktop and mobile devices, as our fuzzing targets. Table 1 provides an overview of the GPU drivers used in our experiments:

- **Nvidia**: We use the Nvidia open-gpu-kernel-modules [19], version 560.35.03. These open-source kernel modules allow us to integrate essential fuzzing instrumentation, such as KCOV and KASAN. For user-space libraries, we use CUDA v12.6.1, Vulkan v1.3.239 and OpenCL v3.0 to fuzz Nvidia GPU drivers.
- **AMD**: We select the AMD ROCK-Kernel-Driver [25], version rocm-6.2.4, as the fuzzing target. Although AMD maintains an on-tree GPU driver in the Linux kernel, the ROCK-Kernel-Driver is the official upstream repository for this driver, offering a stable codebase. We use Vulkan v1.3.239 and OpenCL v3.0 to fuzz AMD GPU drivers.
- **Mali**: We use the Pixel 6 with Android 13 and kernel version v5.10.157, built from the android-gs-raviole-5.10-android13-qpr3 manifest. We use Vulkan v1.3.239 and OpenGL ES 3.2 as the libraries to fuzz it.

Table 1: Experiment Setup

Manufacture	GPU	Kernel Driver	User-Space Libraries
Nvidia	RTX 6000 ADA	open-gpu-kernel-modules (version 560.35.03)	libcuda.so libvulkan.so libnvidia-opencl.so
AMD	Radeon RX 7900 XT	ROCK-Kernel-Driver (version rocm-6.2.4)	libOpenCL.so libvulkan.so
Mali	Mali-G78 MP20	Pixel kernel v5.10.157	libvulkan.so libGLESv3.so

Description generation overhead Generating a description for a single API takes between 30 seconds and 2 minutes, depending primarily on the number of arguments and the complexity of the involved data structures. Generating a complete set of descriptions for an entire library takes between 2 hours (e.g., OpenCL) and 7 hours (e.g., CUDA), depending on the number of APIs. We conducted several fuzzing experiments to evaluate the effectiveness of StepStone-generated descriptions

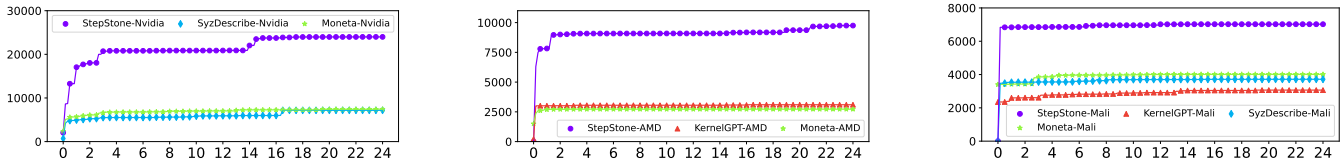
Experiment Environment. Due to hardware limitations, we conducted our experiments on two separate machines:

- For Nvidia GPU, we fuzz it on a machine equipped with 4× AMD EPYC 7543 32-Core Processors and 8× Nvidia RTX 6000 ADA GPUs.
- For AMD GPU, we fuzz it on a machine with AMD Ryzen 7 5800X 8-Core CPU and a Radeon RX 7900 XT GPU.
- For Mali GPU, we fuzz it on Pixel 6 devices.

We enabled KCOV to collect coverage feedback and KASAN sanitizer to capture memory errors.

Experiment Overview. We conducted two fuzzing experiments to evaluate the effectiveness and bug discovery capability of StepStone, and a manual description revision session to evaluate the accuracy of the generated library API description:

❶ **Comparison with other kernel fuzzers.** We've chosen three state-of-the-art kernel fuzzers for this comparison: SyzDescribe, KernelGPT, and Moneta. SyzDescribe is the state-of-the-art static-analysis-based syscall description generator. KernelGPT is the state-of-the-art LLM-based syscall description generator. The comparison with these two solutions is to demonstrate the advantage of using library APIs over syscalls. We chose Moneta because it targets GPU driver fuzzing (including Nvidia, AMD, and Mali).



(a) Nvidia (b) AMD (c) Mali

Figure 11: Comparison of the three GPU fuzzing experiments. The X-axis represents time in hours. The Y-axis represents the number of covered basic blocks.

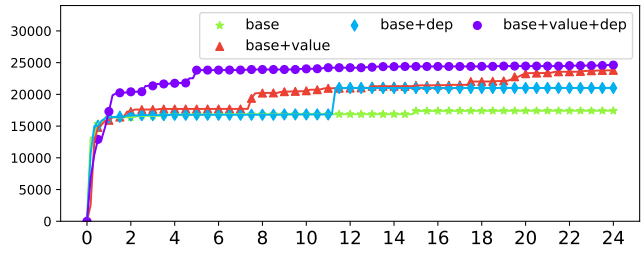


Figure 12: Ablation study results

Specifically, the tool is released with a set of syscall descriptions that we can directly use for fuzzing the GPU drivers. Note that we do not use Moneta's full record-and-replay capability [43]. This is because our goal is to isolate the impact of syzkaller descriptions. After all, StepStone and Moneta's record-and-replay are complementary: StepStone-generated descriptions can be integrated into Moneta. The fuzzing sessions for Nvidia, AMD, and Mali ran for a total of 24 hours, and their total code coverage is compared.

❷ **Bug Discovery.** To demonstrate the real-world impact of StepStone, we conducted three independent 7-day fuzzing sessions using StepStone against Nvidia, AMD, and Mali. The goal was to identify previously unknown GPU driver vulnerabilities and demonstrate the fuzzer's capability in bug discovery.

❸ **Ablation Study.** We conducted an ablation study to evaluate the coverage impact of the proposed LLM components. Specifically, we have `base` components (everything excluding ❸ and ❹ in Figure 7), on top of which we can add `Value Range Recovery LLM` component and the `Dependency Retrieval LLM` component (`value` and `dep` in short). We prepared four control groups of syzkaller descriptions for CUDA: (1) `base`, (2) `base+dep`, (3) `base+value`, and (4) `base+value+dep`. We then ran a 24-hour fuzzing experiment with each description to compare their code coverage. ❹ **Library API Description Verification.** To understand the quality of the generated API descriptions, we manually evaluate the correctness of the generated API descriptions in terms of "API signature", "structure definition", "value range recovery" and "dependency recovery". Specifically, we sampled 50 APIs from each library and the verification was conducted by four researchers with prior experience in syzkaller descriptions by comparing the descriptions against the library documentation.

LLM Cost & Errors StepStone costs on average $1.3 per library API description when using OpenAI o1 (the best model available at the time) and in total $1,200 for the APIs in all four libraries. With newer models (e.g., GPT-5.2), which are both stronger and approximately 6× cheaper, the cost could drop to about $0.2 per API, $200 in total. Although StepStone's `Syzlang Translate Engine` mitigates most syntax errors, we still observe occasional semantic and minor syntax errors (roughly 1 error for every 10 APIs) in the generated descriptions. Many of these minor errors are duplicated cross different descriptions and can be easily fixed using the same method. For example, duplicate `resource` names across multiple descriptions may create conflicts, simply remove the duplicate resource declarations in other descriptions. Structures or function arguments may reference missing types due to LLM context exhaustion. We consider that the LLM fails to extract such types and replace the missing types with `void`. These issues can be mitigated through the use of more advanced models with stronger reasoning capabilities and larger context windows, or improving `Syzlang Translate Engine`, though this increases token cost and generation time. In practice, such syntax and semantic errors are straightforward to fix with minimal manual effort (seconds to minutes per error). Once the descriptions pass compilation, we use them in the subsequent experiments.

7.2. Experiment Results

Comparison with other kernel fuzzers. To evaluate the fuzzing performance of StepStone, we report the code coverage comparison. As discussed in §3.1, SyzDescribe failed to generate descriptions for AMD GPU drivers due to the lack of modeling of `drm_ioctl` function used by the `amdgpu` module. Unfortunately, KernelGPT failed to recover a significant portion of NVIDIA's data structures and type definitions, rendering the generated descriptions irreparable with reasonable effort. Specifically, we observed that the descriptions generated by KernelGPT are highly incomplete and cannot be compiled with Syzkaller beyond easily fixable syntax issues. In particular, the generated syscalls reference over 60 missing structures or type definitions from the NVIDIA GPU drivers. Manually adding all of them requires significant effort, and we consider such descriptions unusable. Addressing this limitation would require significant effort, which lies outside the scope of this

paper. As a result, SyzDescribe will not fuzz against AMD, and KernelGPT will not fuzz against Nvidia.

Figure 11 shows that StepStone achieved superior code coverage at the beginning of the fuzzing session. Within the first hour, StepStone had already surpassed all other fuzzers in the Nvidia GPU, covering over three times as many basic blocks by the two-hour mark. This performance advantage persisted throughout the fuzzing session. Interestingly, SyzDescribe and Moneta cover even less basic blocks in Nvidia than StepStone covers in AMD GPU, despite the Nvidia codebase being 5 times bigger than AMD codebase. Notably, a significant spike in StepStone's coverage occurred at the 14th hour, with an additional 3,000 basic blocks covered, whereas SyzDescribe and Moneta struggled to surpass 10,000 basic blocks by the end of the experiment.

As for AMD drivers, Moneta and KernelGPT achieve nearly identical coverage, while StepStone outperforms both of them by twice the coverage. As mentioned previously, SyzDescribe failed to generate any descriptions for the AMD driver and thus does not have any results.

For Mali drivers, KernelGPT, Moneta, and SyzDescribe all performed worse than StepStone. Notably, StepStone outperformed the best-performing baseline, i.e., Moneta, by achieving roughly 2 times the coverage at the end of the 24-hour period.

Bug Discovery. During the 7-day fuzzing sessions, we fuzzed the libraries using the generated API descriptions for CUDA, Vulkan, and OpenCL. We discovered 11 previously unknown bugs. All of the 11 new bugs come from the Nvidia GPU driver (a subset of them has received CVEs). We suspect this is due to the significantly larger codebase of Nvidia GPU drivers compared to AMD's and Mali's (1.25 million vs. 0.25 million vs. 0.09 million lines of code), with much of Nvidia's code dedicated to CUDA-specific functionality.

Table 2 shows the bug distribution by its type and triggered API category. A total of 7 Nvidia GPU driver bugs were found through fuzzing with CUDA APIs. 2 of them are general protection fault, 1 null-pointer-dereference bug, 3 use-after-free bugs, and an out-of-bounds access bug. Around 64% of the newly found bugs were triggered through CUDA APIs, indicating that the CUDA functionalities in Nvidia GPU driver might undergo rapid development and suffer from many security issues. A total of 4 bugs were discovered in Nvidia GPU driver through Vulkan APIs. One of them is a use-after-free bug, and 3 of them are Warning bugs. We noticed that some bugs cannot be discovered by existing state-of-the-art descriptions, such as those generated by SyzDescribe, because SyzDescribe fails to model certain system calls. We explain this in detail in §7.3.

Regarding the concentration of bugs in a single vendor, this is primarily explained by ❶ codebase size disparity, ❷ functionality concentration (especially CUDA), and ❸ experimental hardware asymmetry.

After a rough estimate of LOC (lines of code) breakdown per feature, we found Nvidia's compute component (CUDA, ∼113k LOC) is ∼3× larger than AMD's (∼45k LOC) and ∼7.5× larger than Mali's (∼15k LOC). The display and infrastructure portion is even more disproportionate (∼750k in Nvidia v.s. ∼144k and ∼55k in AMD and Mali respectively). 8 of the 11 discovered bugs are CUDA-related and three are Vulkan-related, aligning with Nvidia's larger compute and infrastructure components. Nvidia also uniquely exposes a sophisticated CUDA-specific API interface, expanding the fuzzing surface significantly compared to AMD and Mali. Finally, note that our experimental environments were asymmetric due to hardware availability. As mentioned in §7.1, Nvidia fuzzing ran on a multi-CPU server with 8 GPUs, while AMD used a single consumer-grade CPU & GPU. This gives Nvidia's more fuzzing throughput. However, StepStone does not aim to compare vendor security posture; multi-vendor evaluation demonstrates methodological generality rather than ranking security quality.

Table 2: Bug Discovery Results

Library	Bug Type	Bug Title
libcuda	GPF	general protection fault in devmem_page_to_chunk_locked
	GPF	general protection fault in uvm_mmu_chunk_unmap
	NPD	KASAN: null-ptr-deref Read in uvm_page_mask_test
	OOB	KASAN: global-out-of-bounds Read in portMemCmp
	UAF	KASAN: slab-use-after-free Read in block_phys_page_address
	UAF	KASAN: slab-use-after-free Read in mapFind_IMPL
	UAF	KASAN: slab-use-after-free Read in uvm_gpu_chunk_get_gpu
	UAF	KASAN: slab-use-after-free Read in nvidia_mmap_helper
libvulkan	WARN	WARNING in remove_proc_entry
	WARN	WARNING: NVKMS Assert ClearApiHeadStateOneDisp()
	WARN	WARNING: NVKMS Assert AllocDpys()

Ablation Study. We ran each control group's fuzzing experiment for 24 hours, and their coverage results are shown in Figure 12. As expected, the base solution achieved the lowest coverage, slightly above 15,000 basic blocks after 24 hours, whereas the fully enabled configuration (`base+value+dep`) reached this coverage within about one hour. The `base+dep` setup achieved the second-lowest coverage, reaching around 20,000 basic blocks after 24 hours. Although this description benefited from dependency information that enabled more system calls and data connections, its lack of accurate value ranges limited the code coverage. Conversely, the `base+val` setup achieved better coverage than `base+dep`, suggesting that recovering precise value ranges contributes more to the overall coverage than recovering dependencies. This aligns with the fact that the number of arguments and structure fields requiring value recovery far exceeds the number of inter-API dependencies. Finally, the fully enabled description achieved the highest coverage at about 25,000 basic blocks, and continued to grow steadily over time. These results demonstrate the effectiveness of each LLM component and their collaborative impact on improving overall fuzzing coverage.

Library API Description Verification. We randomly sampled 50 APIs each from CUDA, OpenCL, Vulkan and OpenGL ES to evaluate the accuracy of their generated descriptions, with results summarized in Table 3. For "API Signatures", we verified the number and type of its arguments and return value against official documentation. We found only 0.3% and 0.87% APIs in CUDA and OpenCL contain incorrect number of arguments. Note that our parser already extracted the correct API signatures from header files. These errors are likely due to LLM occasional mistakes when con-

Table 3: Description Error Rate

Library	API Signature		Structure	Value Range	Dependency
	Sig Number	Sig Type	Definition		
libcuda	0.3%	4.23%	0.91%	1.51%	0%
libOpenCL	0.87%	6.09%	0%	2.61%	0.87%
libvulkan	0%	3.57%	0.51%	2.04%	0.51%
libGLESv3	0%	5.4%	0%	0%	0.21%

verting them into the Syzlang format. After rerunning these APIs with StepStone, it can return correct number of arguments, supporting our hypothesis. Additionally, we observed 4.23%, 6.09%, 3.57% and 5.4% type error rate in CUDA, OpenCL Vulkan and OpenGL ES, respectively. These errors come from different reasons, such as unsupported C type like `float`, which are not natively handled by Syzlang. Without specific instructions, the LLM might substitute unsupported types with arbitrary Syzlang-supported types like `int32` or pointer types. Additionally, missing type definitions (e.g., due to our code parser errors) sometimes led the LLM to use `intptr` or `int64` to represent a generic pointer. For "Structure Definitions", error rates were low, with 0.91% in CUDA and 0.51% in Vulkan, reflecting the overall accuracy of our code parser. As for "Value Range Recovery", we measured 1.51%, 2.61%, 2.04% APIs still have inaccurate value range. This is mostly due to the unconventional flag definition. Such as using macros (`#define`) to define each flag instead of enclosing them into an enum type. This makes our "Value Range Recovery LLM" fail to recover the actual flag type from the primitive-type flag. Finally, for "Dependencies Recovery", 0.87%, 0.51% and 0.21% of APIs were missing dependencies. These issues are largely due to vague explanation in documentation that often rely on domain knowledge to infer dependencies. Overall, most of these issues can be improved by either adding additional knowledge about Syzlang, improving our implementation, and better prompts.

7.3. Case Study

Figure 13 demonstrates one of our newly found bugs: `KASAN: slab-use-after-free Read in nvidia_mmap_helper`. This UAF read bug was triggered by `mmap` function when the Nvidia GPU driver tries to map kernel pages to user space so the user-space applications can access device memory. However, these kernel pages could already be freed by `nvos_free_alloc()`, but somehow the dangling pointer was still stored inside `mmap_context->alloc`. During `mmap()`, the process eventually invokes `nvidia_mmap_helper` where the device driver's physical pages will be mapped to user space. Line 4 shows that `at` received the dangling pointer from `mmap_context->alloc`, and line 6 triggers the use-after-free read when `at` tries to dereference a data pointer `num_pages`. KASAN captured this UAF read and reported the bug. However, it is a well-known issue that the first bug report does not represent the most critical impact of a bug [56]. If we look further, we found another data pointer dereference from the freed object at line 11.

```
1  int nvidia_mmap_helper(...)
2  {
3      nv_alloc_t *at;
4      at = mmap_context->alloc; // Dangling pointer
5
6      if ((page_index + pages) > at->num_pages) // UAF Read
7      {
8          return -ERANGE;
9      }
10
11     start = at->page_table[page_index]->phys_addr;
12     size = pages * PAGE_SIZE;
13
14     ret = nv_io_remap_page_range(vma, start, size, 0);
15 }
16
17 static inline int nv_io_remap_page_range(struct
       vm_area_struct *vma, NvU64 phys_addr, NvU64 size,
       ...) {
18     ret = remap_pfn_range(vma, vma->vm_start, (phys_addr
           >> PAGE_SHIFT), size, ...); // Page mapping
19 }
```

Figure 13: slab-use-after-free Read in nvidia_mmap_helper

This time, a page pointer was retrieved from the freed object. Because the object `at` points to was freed, an attacker can spray certain objects via heap fengshui [54] to overwrite the original content on the freed object with attacker-controlled data. Therefore, the value of the page physical address `start` can be controlled by the attacker. Notably, physical pages corresponding to this physical address are later mapped into user space through `remap_pfn_range()` (line 18). This means the attacker can potentially map any physical page into user space including sensitive data or even code section, allowing the attacker to access arbitrary physical memory. Interestingly, this bug cannot be found by SyzDescribe because it only models `open()` and `ioctl()`, whereas the bug is triggered through `mmap()`. To support discovering such bugs, SyzDescribe would need to add support for `mmap()`, which involves complicated memory management mechanisms such as page allocation and deallocation. In contrast, StepStone handles this naturally by enabling memory-management-related API calls without requiring detailed modeling of their internal implementations.

7.4. Vulnerability Disclosure

StepStone discovered 11 previously unknown bugs in Nvidia GPU drivers. We reported these bugs to Nvidia and received confirmation that they are actively reviewing and working on our reports. In compliance with Nvidia's Coordinated Vulnerability Disclosure plan, we have agreed not to disclose the bug details to the public until April.

8. Related Work

LLM-Based Bug Finding. Recently, LLMs have emerged as a prominent research topic in the field of bug detection. Li [46] presented a LLM-enhanced static analysis approach to detect UBI (use-before-initialization) bugs. Fuzz4all [51] leverages LLM to provide insights for seed generation and

mutation during fuzzing. TitanFuzz [35] and FuzzGPT [36] target Deep Learning libraries, using LLM to learn all the language and DL computation constraints and synthesize unusual programs for fuzzing. WhiteFox [52] uses LLMs with source-code information to test compiler optimization. KernelGPT [53] uses LLM to assist description generation, which was only written by humans or generated by program analysis. DeepSURF [29] uses LLM to detect memory safety vulnerabilities in Rust.

Kernel Device Driver Fuzzing. DIFUZE [34] presents a static analysis approach to recover device driver interfaces. SyzDescribe [39] further improved the static analysis by modeling device driver operations (e.g., ioctl) to reach a better recovery precision. KSG [50] adopts a dynamic approach using symbolic tracing to collect path constraints, but it suffers from scalability issues on larger device drivers. MoonShine [47] inspects traces of existing test suites to obtain dependencies for seed generation. SyzGen [33] infers the data dependencies and value constraints from other program execution logs. SyzGenPlusPlus [32] analyzes the insertion and lookup operations to recover the device driver dependencies. Moneta [43] proposed an ex-vivo approach to conduct stateful and parallel fuzzing on GPU drivers via a combination of techniques, including snapshot-and-rehost and record-and-replay. This allowed them to fuzz more efficiently and deeper into the drivers. We consider this design orthogonal to StepStone.

User-Space Library Fuzzing. User-space libraries provide new attack surfaces to user-space programs. FuzzBuilder [42] provides an automated fuzzing solution for C/C++ libraries. TLS-Attacker [49] presents multi-stage fuzzing approach to evaluate TLS server behavior. TitanFuzz [35] and FuzzGPT [36] are targeting deep-learning library using the assistant of Large Language Model to synthesize unusual programs as input. FuzzGen [41] builds dependency graphs to obtain data dependencies, and use them in a universal library fuzzing. Hopper [31], on the other hand, does not require library source code. It recovers the fuzzing interface from the library header, and infer dependencies during the fuzzing. Diane [48] and IoTFuzzer [30] use the similar approach of fuzzing external components (companion apps) to expose vulnerabilities in a more complicated system (firmware).

9. Discussions & Limitation

Other GPU & Libraries. Researchers can easily extend StepStone to other GPUs and libraries, despite this paper primarily discussing Nvidia, AMD, and Mali due to their dominant prevalence. There are several other GPUs and GPU libraries that were not discussed in the paper. Intel has a long history of building integrated GPUs, but recently has released its discrete GPUs-Intel Arc [14]. Apple also developed its own GPUs for M-series chips. As for GPU libraries, Direct3D [11] is primarily used for rendering 3D graphics on Windows platforms. Metal [3] is a computing and rendering library for macOS. TensorFlow [27] and PyTorch [23] are deep learning libraries that utilize GPU for parallel computing.

Restriction of Inputs Imposed by Libraries. Libraries may implement sanity checks for critical values; a failed sanity check makes the program return earlier, leading to incomplete code coverage in the device driver. We could not easily assess the number of such sanity checks due to the closed-source nature of GPU libraries, and Nvidia prohibits reverse engineering its library binary [6]. However, the impact of sanity checks in kernel fuzzing may not be that significant. We have seen many Linux kernel CVEs [16] solely rely on libraries (e.g., `libnfnetlink`) in their exploits. Overall, we'd like to clarify that StepStone is not intended to replace syscall-level fuzzing. Rather, it complements syscall-fuzzing by enabling semantically valid, documentation-informed exploration in cases where syscall modeling is challenging/impractical (e.g., heavily overloaded ioctl paths in GPU drivers). Despite the potential false negatives, StepStone still found 11 new GPU driver bugs, proving its bug discovery capability. We envision that future work could be done to mitigate the sanity check issue. As for StepStone, we believe our approach is a concrete step forward in exploring GPU kernel driver fuzzing.

10. Conclusion

In this paper, we presented StepStone, a novel kernel fuzzer that fuzzes user-space GPU libraries to indirectly fuzz the GPU kernel driver, utilizing the LLM-generated library description. StepStone demonstrates its efficiency by finding in total of 11 previously unknown bugs in Nvidia GPU drivers.

11. Ethics Considerations

StepStone discovered 11 previously unknown bugs in Nvidia GPU drivers. We reported these bugs to Nvidia and received confirmation that they are actively reviewing our reports. To date, a subset of these vulnerabilities has received CVEs. In compliance with Nvidia's Coordinated Vulnerability Disclosure plan, we have agreed not to disclose the bug details to the public until later in the year. However, the submission cycle timeline allows us to safely include the bug details in the case study.

12. Open Science

We will open-source StepStone and its related resources once the paper is accepted, including the source code, LLM instructions and prompts, as well as the generated descriptions for CUDA, OpenCL, Vulkan, and OpenGL.

13. Acknowledgment

We thank the anonymous reviewers for their insightful comments and valuable suggestions. This material is based upon work supported by the National Science Foundation under Grant No. #2452292 & #2247881 and the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112590041.

Appendix A. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

This paper presents StepStone, a GPU driver fuzzing framework that leverages user-space GPU libraries as the fuzzing interface instead of directly fuzzing system calls. StepStone uses LLMs to analyze library documentation and automatically generate syzkaller-compatible interface descriptions, enabling more semantically valid fuzzing inputs.

A.2. Scientific Contributions

- Creates a New Tool to Enable Future Science.
- Identifies an Impactful Vulnerability.
- Provides a Valuable Step Forward in an Established Field.

A.3. Reasons for Acceptance

1) **Provides a valuable step forward in an established field.** The paper proposes fuzzing GPU drivers through user-space libraries, which provide richer semantics and documentation compared to raw system calls. Reviewers agreed that applying this idea to GPU driver fuzzing is novel and promising.
2) **Creates a new tool to enable future science.** StepStone integrates LLM-assisted interface description generation with a syzkaller-based fuzzing pipeline and demonstrates improved coverage compared to prior approaches.
3) **Identifies impactful vulnerabilities.** The system discovered previously unknown bugs in GPU drivers, demonstrating the effectiveness of the approach.

References

[1] Opengl. https://www.opengl.org/, 2018.

[2] Amd cpu iommu. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/specifications/48882_IOMMU.pdf, 2024.

[3] Apple metal. https://developer.apple.com/metal/, 2024.

[4] clenqueuesvmmigratemem. https://registry.khronos.org/OpenCL/sdk/3.0/docs/man/html/clEnqueueSVMMigrateMem.html, 2024.

[5] Cuda. https://developer.nvidia.com/cuda-toolkit, 2024.

[6] Cuda copyright and licenses. https://docs.nvidia.com/compute-sanitizer/CopyrightAndLicenses/index.html, 2024.

[7] Cve-2021-39793. https://nvd.nist.gov/vuln/detail/cve-2021-39793, 2024.

[8] Cve-2022-38181. https://nvd.nist.gov/vuln/detail/cve-2022-38181, 2024.

[9] Cve-2023-6241. https://github.blog/security/vulnerability-research/gaining-kernel-code-execution-on-an-mte-enabled-pixel-8/, 2024.

[10] Cve-2024-36974. https://ssd-disclosure.com/ssd-advisory-linux-kernel-taprio-oob/, 2024.

[11] Direct3d. https://learn.microsoft.com/en-us/windows/win32/direct3d, 2024.

[12] File search. https://platform.openai.com/docs/assistants/tools/file-search, 2024.

[13] Hopper github issue. https://github.com/FuzzAnything/Hopper/issues/17, 2024.

[14] Intel arc. https://www.intel.com/content/www/us/en/products/details/discrete-gpus/arc.html, 2024.

[15] libfuzzer. https://releases.llvm.org/8.0.1/docs/LibFuzzer.html, 2024.

[16] Liunx kernel cves. https://github.com/star-sg/CVE, 2024.

[17] Mali gpu. https://developer.arm.com/documentation/102849/latest/, 2024.

[18] Nvidia cves. https://www.nvidia.com/en-us/product-security/, 2024.

[19] open-gpu-kernel-modules. https://github.com/NVIDIA/open-gpu-kernel-modules, 2024.

[20] Openai assistant. https://platform.openai.com/docs/assistants/overview, 2024.

[21] Opencl. https://www.khronos.org/opencl/, 2024.

[22] Opengl. https://www.vulkan.org/, 2024.

[23] Pytorch. https://pytorch.org/, 2024.

[24] Retrieval augmented generation (rag). https://help.openai.com/en/articles/8868588-retrieval-augmented-generation-rag-and-semantic-search-for-gpts, 2024.

[25] rock-kernel-driver. https://github.com/ROCm/ROCK-Kernel-Driver, 2024.

[26] syzlang. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md, 2024.

[27] Tensorflow. https://www.tensorflow.org/, 2024.

[28] vfio. https://docs.kernel.org/driver-api/vfio.html, 2024.

[29] Georgios Androutsopoulos and Antonio Bianchi. deepsurf: Detecting memory safety vulnerabilities in rust through fuzzing llm-augmented harnesses. *arXiv preprint arXiv:2506.15648*, 2025.

[30] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*, pages 1–15, 2018.

[31] Peng Chen, Yuxuan Xie, Yunlong Lyu, Yuxiao Wang, and Hao Chen. Hopper: Interpretative fuzzing for libraries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1600–1614, New York, NY, USA, 2023. Association for Computing Machinery.

[32] Weiteng Chen, Yu Hao, Zheng Zhang, Xiaochen Zou, Dhilung Kirat, Shachee Mishra, Douglas Lee Schales, Jiyong Jang, and Zhiyun Qian. Syzgen++: Dependency inference for augmenting kernel driver fuzzing. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, pages 4661–4677. IEEE, 2024.

[33] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. Syzgen: Automated generation of syscall specification of closed-source macos drivers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 749–763, New York, NY, USA, 2021. Association for Computing Machinery.

[34] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2123–2138, New York, NY, USA, 2017. Association for Computing Machinery.

[35] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 423–435, New York, NY, USA, 2023. Association for Computing Machinery.

[36] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[37] Natanael Fraga. Challenging llms beyond information retrieval: Reasoning degradation with long context windows. 2024.

[38] Google. syzbot. https://syzkaller.appspot.com/upstream/, 2020.

[39] Yu Hao, Guoren Li, Xiaochen Zou, Weiteng Chen, Shitong Zhu, Zhiyun Qian, and Ardalan Amiri Sani. Syzdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers. In *44rd IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 22-25, 2023*. IEEE, 2023.

[40] Yu Hao, Hang Zhang, Guoren Li, Xingyun Du, Zhiyun Qian, and Ardalan Amiri Sani. Demystifying the dependency challenge in kernel fuzzing. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 659–671, New York, NY, USA, 2022. Association for Computing Machinery.

[41] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. FuzzGen: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2271–2287. USENIX Association, August 2020.

[42] Joonun Jang and Huy Kang Kim. Fuzzbuilder: automated building greybox fuzzing environment for c/c++ library. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, page 627–637, New York, NY, USA, 2019. Association for Computing Machinery.

[43] Joonkyo Jung, Jisoo Jang, Yongwan Jo, Jonas Vinck, Alexios Voulimeneas, Stijn Volckaert, and Dokyung Song. Moneta: Ex-vivo GPU driver fuzzing by recalling in-vivo execution states. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2025.

[44] Ummay Kulsum, Haotian Zhu, Bowen Xu, and Marcelo d'Amorim. A case study of llm for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, AIware 2024, page 103–111, New York, NY, USA, 2024. Association for Computing Machinery.

[45] Guoren Li, Hang Zhang, Jinmeng Zhou, Wenbo Shen, Yulei Sui, and Zhiyun Qian. A hybrid alias analysis and its application to global variable protection in the linux kernel. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4211–4228, Anaheim, CA, August 2023. USENIX Association.

[46] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proc. ACM Program. Lang.*, 8(OOPSLA1), April 2024.

[47] Shankara Pailoor, Andrew Aday, and Suman Jana. MoonShine: Optimizing OS fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, Baltimore, MD, August 2018. USENIX Association.

[48] Nilo Redini, Andrea Continella, Dipanjan Das, Giulio De Pasquale, Noah Spahn, Aravind Machiry, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for iot devices. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 484–500, 2021.

[49] Juraj Somorovsky. Systematic fuzzing and testing of tls libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1492–1504, New York, NY, USA, 2016. Association for Computing Machinery.

[50] Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. KSG: Augmenting kernel fuzzing with system call specification generation. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 351–366, Carlsbad, CA, July 2022. USENIX Association.

[51] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[52] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. Whitefox: White-box compiler fuzzing empowered by large language models. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024.

[53] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. Kernelgpt: Enhanced kernel fuzzing via large language models. ASPLOS '25, page 560–573, New York, NY, USA, 2025. Association for Computing Machinery.

[54] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. Playing for K(H)eaps: Understanding and improving linux kernel exploit reliability. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 71–88, Boston, MA, August 2022. USENIX Association.

[55] Hang Zhang, Weiteng Chen, Yu Hao, Guoren Li, Yizhuo Zhai, Xiaochen Zou, and Zhiyun Qian. Statically discovering high-order taint style vulnerabilities in os kernels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 811–824, New York, NY, USA, 2021. Association for Computing Machinery.

[56] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. SyzScope: Revealing High-Risk security impacts of Fuzzer-Exposed bugs in linux kernel. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3201–3217, Boston, MA, August 2022. USENIX Association.

Appendix

Your task is extracting the enum type of integer flag arguments/fields in a given API or data structure.
An integer flag argument/field is a variable that does not clearly indicate its potential values or data type (e.g., int flags or long flags). You should always refer to the uploaded API documentation in your file search to infer the data type of the flag argument.

I will ask you to count how many integer flag arguments/fields are in the given API or data structure.
- Your response should be a JSON object with the following format: {”integer_flags”: [{”type”: [”flag_type”], ”explain”: ”explanation”}]}
- Your explanation should written in natural language, describing which integer flag should be what types.
- If there are no flag arguments, return an empty flags list as {”integer_flags”: []}

Here I provided 4 examples for you to learn.

Figure 14: Instruction for Flag Retrieval LLM

Task: Your objective is to identify arguments or return values in an API that should be considered “resources” based on their use as inputs or outputs in other APIs. You should only refer to the provided API documentation for this task.

Instructions:
1. Check Pointer Types:
- Look at pointer types in both arguments and return values of the given API.
- Determine if any other APIs in the documentation use these as input values or produce them as output values.

2. Identify Resources for Cross-Type Pairing:
- Focus only on arguments or return values that differ in type from the corresponding arguments or return values in the current API.
- Exclude any obvious resources of the same type across APIs.

3. Output Format:
- For each resource found, output a line in natural language in English, such as: ”¡producer argument¿/¡producer return value¿ should be marked as a resource ”¡producer API name¿_¡producer argument¿/¡producer return value¿”, because it is used/produced by ¡API name¿.”
- If there are no dependencies (i.e., no arguments or return values are used by other APIs), respond with: ”NO DEPENDENCY”
- When respond with ”NO DEPENDENCY”, your response should only contain the ”NO DEPENDENCY” string, no explanation is needed.

Here I provided 2 examples for you to learn

Figure 15: Instruction for Dependency Retrieval LLM


```
{
   "structure_name": "CUexecAffinityParam",
   "fields": [
     {
       "field_name": "type",
       "field_type": {"type_name": "flags", "flags_enum": "CUexecAffinityType", "flags_element_size": "int32"}
     },
     {
       "field_name": "param",
       "field_type": {"type_name": "arg", "arg_name": "CUexecAffinityParam_OP1"}
     }
   ],
   "type_cast": null,
   "arg_of": []
 }
```


Figure 16: CUexecAffinityParam JSON Representations

```
{"type_name": "const", "const_value": "", "const_size": ""}

{"type_name": "int", "int_bound": {"min_val": "", "max_val": ""}, "int_size": ""}

{"type_name": "flags", "flags_enum": "", "flags_element_size": ""}

{"type_name": "array", "array_element": "", "array_length": ""}

{"type_name": "ptr", "ptr_method": "", "ptr_object": ""}

{"type_name": "string", "string_value": ""}

{"type_name": "len", "len_target": "", "len_size": ""}

{"type_name": "bytesize", "bytesize_target": "", "bytesize_size": ""}

{"type_name": "bitsize", "bitsize_target": "", "bitsize_size": ""}

{"type_name": "void"}

{"type_name": "structure", "structure_name": "", "args":[]}

{"type_name": "arg", "arg_name": ""}

{"type_name": "resource", "resource_name": ""}
```

Figure 17: Predefined JSON Type Schema